An Incremental Language Conversion Method

to Convert C++ into Ada95

THESIS

Ding-yuan Sheu, Captain, ROC Army
AFIT/GCS/ENG/96D-33

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

An Incremental Language Conversion Method

to Convert C++ into Ada95

THESIS

Ding-yuan Sheu, Captain, ROC Army
AFIT/GCS/ENG/96D-33

AFIT/GCS/ENG/96D-33

An Incremental Language Conversion Method to Convert C++ to Ada95

THESIS

Presented to the faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the Requirements for the Degree of

Master of Science in Computer System

with

Emphasis in Software Engineering

Sheu, Ding-Yuan, B. S.

Captain, ROC Army

December 1996

Acknowledgments

The research that went into this thesis could not have been possible without the contributions of the numerous individuals enumerated below.

First and foremost, I would like to thank my thesis advisor, Major Mark A. Kanko. The initial idea, incremental conversion to explore the mixed language programming is a remarkable idea. I appreciate his guidance and patience for my thesis work. Not only did I gain knowledge from him for this thesis work, I also learned some American virtues.

I would also like to thank Doctor Patricia Lawlis and Major Keith Shomper for being members of my thesis committee. Doctor Lawlis gave me invaluable opinions for my thesis. Major Shomper always gave another view to solve the problems and difficulties.

Doctor Cyrille Comar, in GNAT at New York University, patiently answered all my silly questions about the GNAT C++ low-level interface. Mr. Tom Quiggle at Silicon Graphics, Inc. (SGI) provided the Adabindgen to help me do this thesis work and gave me some invaluable knowledge in mixed C++ and Ada programming.

My fellow classmates provided moral support even while I suspected whether I would ever graduate. Captain (Capt.) Shawn Hannan, Capt. Charles Beem, and Capt. Vincent Hibbdon helped me survive AFIT's torture. My first friend in AFIT, Capt. David H. Kaneshiro helped me and proofread my thesis draft. He gave me good advice on my thesis writing. My sponsors, Mr. Cecil Martin and Capt. Charles Wright, and their family make me feel like I was studying in my own country.

The following Taiwanese students and their family gave me support and encouragement when I felt lonesome and homesick. They treated me as a member of their families:

- Major Wang, Wen-Lung (Drake) and his wife Julie, daughter Tina and son William.
- Lieutenant Colonel Hwang, Kungfu (Howard) and his wife Jenny, daughter Jennifer and son Johnathan.
- Major Chiou, Sen-tzer (Daniel) and his wife Lindy.
- Major Wu, Wei-ya (Wayne), and his wife Hsu-yu, daguters Tina and Chelsy.
- Major Ke, Yushen (Kevin) and his wife Juila.

I also appreciate Major Alan Hwang and Colonel Den-Ming Mar. Without their recommendation and help, I would not have this opportunity to study at AFIT. They are two important persons who gave me a big help in my military career.

Working on this thesis took a great deal of time and effort, and for a man who studied abroad in a different language and culture from his home country, my mother and father provided me tremendous spiritual support. I also wish to think appreciate my two younger sisters who took care of our parents while I was studying at AFIT. Bauor-Zen Sheu, the elder one was married in Feb 1996. I wish her and her husband the best of lucks. I also wish my younger sister, Su-How Sheu, the best of luck in starting her family.

There are so many people I need to thank, including some that I do not even know their names. I cannot begin to list all their names here to express my appreciation. The only words I can say are "Thank God". Thank God for giving so many nice people to help me in my life.

Ding-yuan Sheu (Steven)

Oct 15, 1996.

Table of Contents

List of Figures

List of Figures (Cont'd)

List of Tables

Abstract

This thesis develops a methodology to incrementally convert a legacy object-oriented C++ application into Ada95. Using the experience of converting a graphic application, called Remote Debriefing Tool (RDT), in the Graphics Lab of the Air Force Institute of Technology (AFIT), this effort defined a process to convert a C++ application into Ada95.

The methodology consists of five phases: (1) reorganizing the software application, (2) breaking mutual dependencies, (3) creating package specifications to interface the existing C++ classes, (4) converting C++ code into Ada programs, and (5) embellishing. This methodology used the GNAT's C++ low-level interface capabilities to support the incremental conversion. The goal of this methodology is not only to correctly convert C++ code into Ada95, but also to take advantage of Ada's features which support good software engineering principles.

An Incremental Language Conversion Method to Convert C++ to Ada95

1 Introduction

1.1 Overview

This thesis work developed a methodology to incrementally convert a legacy object-oriented C++ application to Ada95. From the experience of converting a graphic application in the Graphics Lab of the Air Force Institute of Technology (AFIT), this research defined a process to translate a C++ application to Ada95. Also, it provided another way to develop new graphics applications in the Graphics Lab of AFIT. Some of the issues investigated during this reaearch included: language conversion process, mixed programming language development, and object-oriented programming in C++ and Ada95.

1.2 Thesis Statement

Even though Ada was the official programming language for the Department of Defense, Major Michael Thurman Gardner chose C++ to develop the Remote Debriefing Tool (RDT) in 1993 [Gardner93]. RDT is a system that utilizes the Distributed Interactive Simulation (DIS) communication protocol to monitor and play back a Red Flag mission. Red Flag was conceived in 1975 to provide realistic simulated air combat missions. Aircrews at Red Flag plan and then execute simulated combat missions. After a simulated mission, all participants have the opportunity to critically review every aspect of their flight

performance and to learn how to improve their flying skills and judgment for future missions in real combat. RDT is a tool that helps all participants reexamine their Red Flag mission performance at remote sites [Gardner93].

One main reason that Major Gardner chose C++ instead of Ada to develop RDT was that the C++ programming language supports object-oriented features. The early Ada language, called Ada83, did not support object-oriented features even though it had many other advantages over C++. Object-oriented features make software easier to reuse. Additionally, the software environment for C++ graphics applications development better supported programmers than did the Ada environment. Therefore, Major Gardner chose C++ as the programming language for RDT.

Ada is a programming language for the complex world. It contains common features of other programming languages and provides additional ability to support complex and large software projects [HBAP].

- Portability: An Ada-developed software system can easily be ported to other operating systems and platforms because Ada is an international standardized software language by MIL-STD-1815A, ANSI, and ISO [HBAP]. Actually, it was the first and is the only existing international standardized object-oriented programming language.

- Modularity: Ada organizes code into self-contained modules such that each software module can be planned, written, compiled, and tested separately. Ada also provides consistency checking across each of these self-contained modules. This allows the software system to be developed by teams and then integrated into a well-structured

system. Therefore, by using Ada, it is easier to develop a software system meeting good software engineering principles.

• Reusability: Ada provides a "package" concept that has advantages over other programming languages. It minimizes the ripple effect when developers retrieve, use and/or change software components, so that it provides an easy way to reuse software components [HBAP]. Additionally, its generic program units allows programmers to perform the same logical function on more than one type of data [HBAP].

• Reliability: Ada's strong-typing, exception handling and tasking features help developers to build reliable software systems [HBAP]. Ada's strong typing enables developers to detect potential bugs in the early development phases instead of later in the development cycle. Some experiments have suggested that strongly typed languages lead to increased program clarity and reliability [Gannon77]. Ada's exception handling mechanism supports developers in building fault-tolerant software. Also, its tasking features help developers to solve real-time problems instead of using lower-level and error-prone operating system calls.

• Maintainability: Ada's modularity and readability make it easier to maintain. Its superior modularity allows programmers to modify a package without affecting other program modules that should not be modified. Ada's readability makes it easier for one programmer to maintain a program written by another programmer [HBAP].

Ada95 revised the early Ada83 such that Ada now has new features to meet current software development requirements. The important new features of Ada95 include three aspects [APR93]:

- Object Oriented Programming: Ada95 has included Object-Oriented Programming (OOP) facilities to allow programmers to do object-oriented programming [APR93]. OOP provides information hiding capability and promotes the reuse of code through the inheritance mechanism [Pohl93].

- Hierarchical Libraries: The hierarchical library form is valuable for the control and decomposition of large software systems [APR93].

- Protected Objects: The enhanced tasking features of Ada95 supports an efficient mechanism for multitask synchronized access to shared data such that the real-time requirement can be easily implemented in Ada95 [APR93].

Since the emergence of Ada95 has eliminated the defects of Ada83 and Ada95 has so many advantages over other languages, it is time to convert old C++ applications to Ada95.

1.3 Scope

This thesis work focused on programming language conversion instead of software reengineering, though some information was obtained by using reverse engineering techniques. This thesis presents the general process of converting one language to another language. Specifically, converting C++ to Ada95 is addressed in this thesis. No new functions or requirements for RDT are introduced in this thesis work. Incremental and partial system conversion is implemented in this thesis in order to obtain immediate results. Also, instead of converting all the software modules implemented in C, an interface was created to interface some existing C-code modules. No hardware issues are considered in

this thesis. The converted Ada95 application should work as well as the original C++ application on the original hardware and software environment.

In order to succeed in the conversion, a number of issues related to this thesis work had to be investigated:

1. Mixed language programming, especially for C/C++ and Ada95.
2. The C++ and Ada95 implementation for object-oriented design.
3. The corresponding data types between C/C++ and Ada95.
4. General language conversion methods.
5. Incremental development methods.

1.4 Assumption

This section enumerates the assumptions that were made in developing the thesis statement and scope of this work.

1. The original software application was built based on an object-oriented design.
2. The original object-oriented design was well-done and no software architecture restructuring is needed.
3. The original application was successfully developed and met the original system requirements.
4. The foundations of RDT including Object Manager, ObjectSim, Performer library and Graphics library were well developed.

1.5 Thesis Goals

There were three main goals in this thesis work:

1. A converted Ada95 RDT. This thesis converted the original C++ RDT application into an Ada95 RDT application without any functional change.

2. A methodology to incrementally convert a C++ application into an Ada95 application. By converting the RDT from C++ into Ada95, this thesis defined a methodology to incrementally convert a C++ application into an Ada95 application. The conversion method and process is addressed in this thesis.

3. A new software development environment for Ada programmers for the Graphics Lab at AFIT. The original software development environment was for C++ programmers. The foundations of software development in the Graphics Lab at AFIT were Performer, Graphics Library (GL), ObjectManager, and ObjectSim. These foundations were created for C++ programmers. After this thesis effort, a new software development environment is available to the Ada programmers.

1.6 Research Approach

Since one goal of this research was to convert a C++ software application, RDT, into Ada95, the first step was to examine and understand the software architecture of RDT. *Software architecture* is the description of a software system with its components and the interactions among these components [Garlan93]. Software architecture has at least five positive impacts in software system development: understanding, reuse, evolution, analysis, and management [Garlan95]. By understanding the software architecture of the

RDT, a big picture view can be attained. That view provides insight into the conversion sequence of the software components.

The second step was to compare the differences between C++ and Ada95. In this step, the parallel data types between C++ and Ada95 and object-oriented implementation in both languages were key issues. These are the foundations of converting one language into another language in an object-oriented architecture.

In order to reduce the risk of converting RDT from C++ into Ada95, the incremental conversion process was chosen. It is easier to identify problems when they occur as the incremental conversion proceeds. By converting one module at a time, problems should be limited to the converted modules, so programmers can focus on the limited areas to identify and solve the problems. Otherwise, it is very difficult to identify problems when they occur. Therefore, the incremental conversion method was chosen for this thesis.

Because this research was going to develop an incremental conversion method, mixed language programming was an inevitable issue. Therefore, the interface between C++ and Ada95 and its capability were investigated in the third step. The capabilities and limitations of multilingual programming play an important role in an incremental conversion method. Without this capability, incremental conversion would not be feasible. The tools to support the conversion process are also very important in this step.

The fourth step was to develop some guidelines that determine the conversion sequence. Good conversion principles can ease and smooth the whole conversion process

and shorten the conversion time. In this step, multilingual software engineering issues should be considered.

The final step was converting one software component at a time, based on the knowledge developed in the previous steps.

1.7 Support

This conversion work was done on Silicon Graphics, Inc. (SGI) machines running the IRIX 5.3 operating system. SGI also provides Graphics Library (GL) and Performer to help users develop their graphics applications on the SGI machines. The new prototype tool, called Adabindgen, also supports developers in interfacing C/C++ modules in Ada programs. Adabindgen will be introduced in a later section. The Static Analyzer helps software engineers understand the structure of legacy software systems and the dependencies among the software modules. The GNAT and SGI's CC compilers were used in this thesis work to compile the Ada95 and C++ software modules, respectively.

1.8 Document Overview

This research investigated developing an incremental language conversion method from C++ to Ada95 and exploring the strength and difficulties of mixed language programming. The goals, assumptions, research approach, and support for this thesis have been outlined throughout this chapter. The remaining chapters describe the research completely.

Chapter 2 details background topics pertinent to this effort. It covers the software layer structure of RDT and the object-oriented model of RDT. The RDT software layer

structure shows how RDT was built and the object-oriented model of RDT shows the key components of RDT and how they associate to each other. These provide the high-level knowledge of RDT.

Chapter 3 summarizes the research related to this thesis work. It begins by presenting some issues of concern in mixed language programming. Then, language conversion methods are reviewed. Finally, two important tools, Adabindgen and GNAT, used in this thesis work are introduced and investigated.

Chapter 4 presents the methodology developed from this thesis work. This methodology suggest five phases to incrementally convert the C++ application into Ada95 by using GNAT's C++ low-level interface capability. These phases are (1) reorganizing the software application, (2) breaking mutual dependencies, (3) creating interface package specifications, (4) converting C++ code into Ada programs, and (5) embellishing.

Chapter 5 presents the problems encountered while trying to achieve the results in this thesis work. Some mixed language lessons have been learned from this thesis work. Mixed language programming has its strengths over single language programming. However, many issues still cause programmers to stumble. These mixed programming difficulties are also examined in Chapter 5.

At a conclusion to this effort, Chapter 6 suggests some areas of future study and work related to this thesis work.

2 Background

The functions and requirements of RDT can be found in [Gardner93]. This chapter introduces the software architecture of RDT.

2.1 Software Layer Structure

In Figure 1, the software layer structure of the RDT application is shown. RDT heavily relies on ObjectSim and Object Manager. Both of them were written in C++. Only C++ programmers can build their applications on ObjectSim and Object Manager. IRIS Performer and GL were supplied by SGI for C++ developers. SGI also supports the Ada binding of Performer and GL such that Ada programmers can use Performer and GL. In order to convert the RDT application from C++ into Ada95, it is necessary to interface ObjectSim and Object Manager.

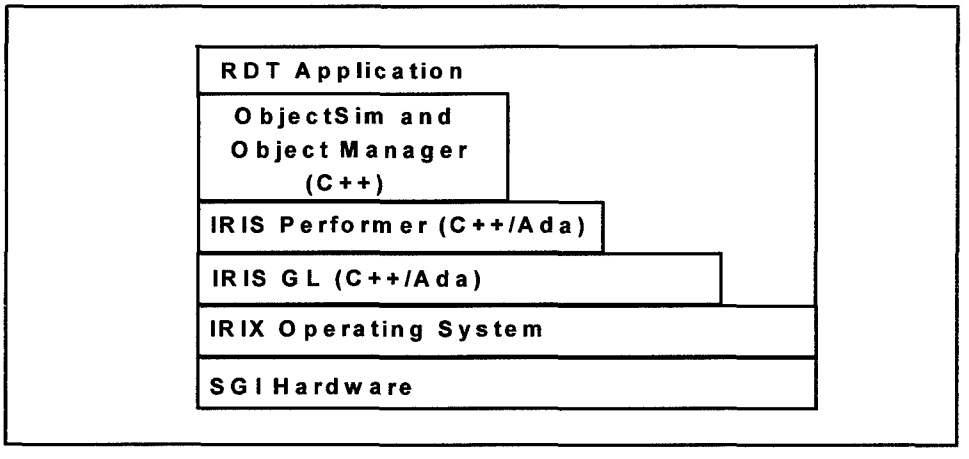


Figure 1. The software layer structure of RDT application.

The IRIS Graphics Library (GL) is a library of low-level graphics subroutines that can be called by programmers to draw and animate 2-D and 3-D color graphics scenes [SGI92]. The GL provides the following capabilities:

- drawing points, lines, and polygons.
- handling of user input.
- animating effects.
- hidden surface removal.
- lighting effects.
- pixel control and setting.
- coordinate transformation.
- frame buffer management.
- antialiasing.
- atmospheric effects.

In contrast to GL, the IRIS Performer Library is an extensible software toolkit that can help developers create real-time 3-D graphics and visual simulation applications [SGI94]. Performer provides the following features:

- building a visual simulation application.
- setting up the display environment.
- building a scene graph.
- database traversal and importing.
- frame and load control.
- creating visual effects.

ObjectSim [Snyder93] provides a set of high-level software components that the programmers in the Graphics Lab of AFIT can reuse to easily build a visual simulation application. The next section will show that the software components of the ObjectSim were successfully reused in the RDT application. Developers can simply develop new software components based on their own unique requirements. Then, developers can construct a whole simulation application by combining the developed software components with the existing ObjectSim reusable software components.

Object Manager is a library which handles a network interface. Object Manager has three main functions: (1) position/time monitoring, (2) dead reckoning, and (3) sending and

receiving the Protocol Data Units (PDU) over the network through interfacing to a low-level driver, called the *network daemon* [Snyder93]. With the help of Object Manager, the distributed visual simulation application can focus on the high level function instead of the low-level network interface issues.

2.2 The Object-Oriented Model of RDT

Figure 2 shows the object-oriented model of RDT using Rumbaugh's [Rumbau91] notation. The classes enclosed by the bold frame belong to ObjectSim and Object Manager. From this figure, readers can form a big picture of the RDT software architecture and understand the relationship among the RDT classes and ObjectSim/Object Manager classes. Most of the RDT classes are derived from ObjectSim. Other RDT classes are built according to unique RDT requirements. That is the success of ObjectSim: build a reusable foundation such that graphic applications can be built based on this reusable infrastructure and only application-unique components have to be built from scratch.

The software modules of RDT and their sizes are listed in Appendix A.

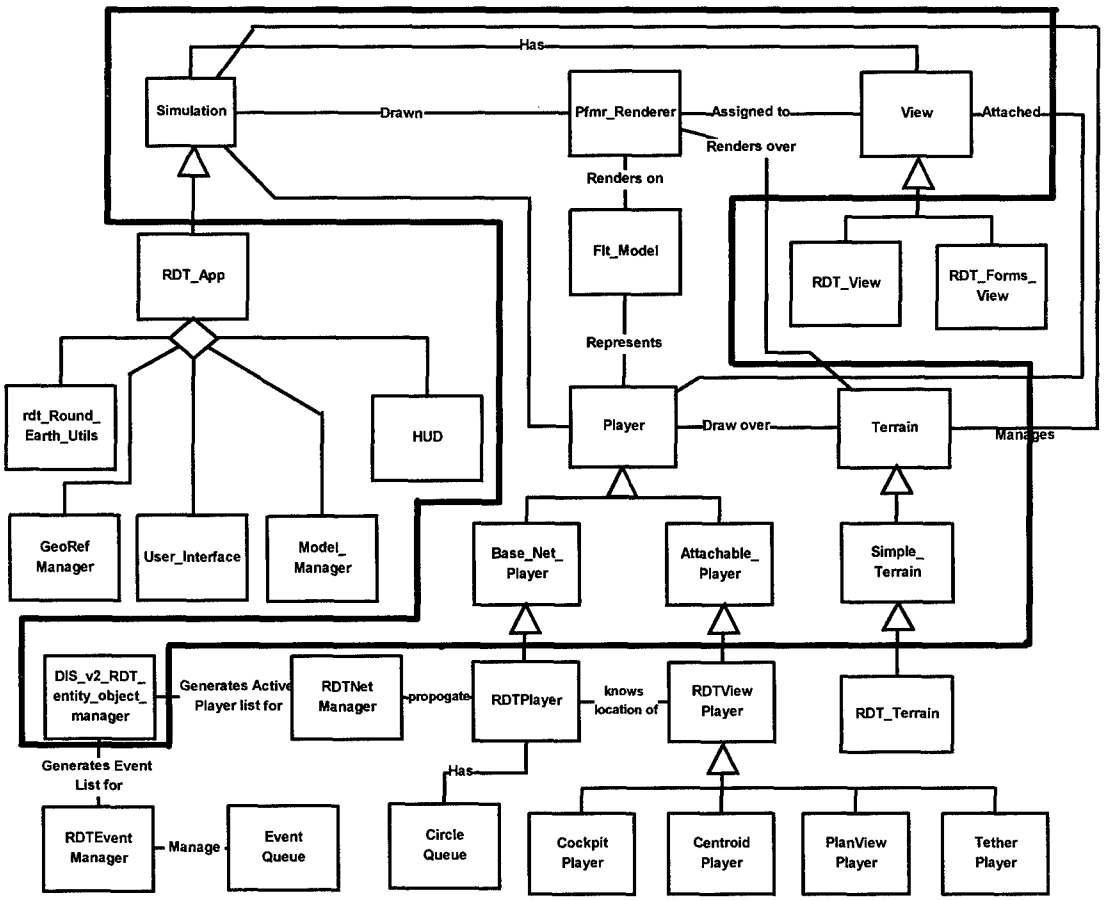


Figure 2. The Object-Oriented Model of RDT

3 Summary of Current Knowledge

3.1 Multilingual Software Engineering

David Hughes roughly classified multilingual software engineering into two categories: reuse of existing software and multilingual implementation by design [Hughes91]. For the reuse of existing software, software engineers need to completely understand the mechanism of the existing code and the language interface mechanism [Hughes91]. That makes the reuse more difficult in multilingual software engineering than single language implementations. For software systems which are multilingual by design, software engineers need to understand the strengths and weaknesses of each programming language, as well as to thoroughly control the effect produced by the languages working together [Hughes91].

David Hughes also pointed out some important areas that needed to be conquered in multilingual software engineering using Ada and C. Most of these issues are general issues not specific to only Ada and C. This section summarizes those issues that are pointed out by [Hughes91] and lists them in decreasing order of importance in the following subsections.

3.1.1 Call Stack Ordering

Every compiler has its own way of handling parameter retrieval from the call stack. Some compilers may use a front-to-back method to process the association of actual arguments popped from the stack with registers or memory locations corresponding to the formal parameters, but some compilers may use the reverse way to do it. When defining an

Ada subprogram interface specification to another language, it may be necessary to reverse the order of the formal arguments. It is not safe to assume that the call interface will be consistent [Hughes91].

Therefore, Hughes reminded multilingual software engineers: "Disassembled listings must be examined early in the design phase in order to establish an accurate paradigm for the language call interface" [Hughes91]. If this problem occurs, a method must be developed to conquer it. In [Kapur95], a method, called "patch code" was introduced to solve this problem. "The patch code adjusts the stack so that the parameters from the calling stack matches the parameters of the function that is being called" [Kapur95].

3.1.2 Compiler Dependencies

Compilers use different ways to implement the return of a single discrete value from a function. C compilers may return the discrete function values via registers instead of the stack. Generally, this won't cause any problem. However, that is not guaranteed. For embedded software systems, it is common to use lower level language programming, such as machine code or assemble language. In this situation, programmers need to manipulate a register directly in order to emulate a function return from C [Hughes91].

3.1.3 Data Representation and Interoperability

"The issue of data type should be addressed at the call interface level" [Hughes91]. Like the idea of Remote Procedure Call, a definition of a single, clearly and sufficiently common abstraction is needed for multilingual software engineering to support the

information exchange from one software module to another different language software module. The data representations of data across languages has to be consistent. *Interoperability* refers to the consistency of data representation across languages and software environments. One approach to achieve interoprability is using so-called language binders. This approach implicitly uses the notion of sharing data structures by abstracting and hiding the essential details of implementation. Therefore, the information exchange relies on the language binder to assure that representation detail is interpreted consistently. In this way, any data structure may coexist in different language systems and can be directly manipulated from any language system [Hughes91].

Complex data structures should be encapsulated in only a single language. If it is necessary to exchange the information of complex data structures among the different language systems, the information should be treated as in black-box fashion and confined to the *safe types*. For example, the black-box datum usually is a pointer to the complex data structure that wants to cross the language boundary [Hughes91]. So complex data structure should be encapsulated in one language, but accessed via pointers from the other language. Also, the data structure must be manipulated in a way that is consistent with the way it was defined.

3.1.4 Parallel Data Types

"*Safe types* are those which map to an underlying representation supported directly by the machine and which have equivalent allowable operation in each of the languages involved" [Hughes91]. For the call interface between C and Ada, there are three safe types:

integer type of some specific size, single-precision floating type, and an address type. Here is an example of safe types for the MC680x0 architecture.

Table 1. The Safe Types of MC680X0 Architecture [Hughes91]

Language	Data Type	Size in Bits
Ada	subtype INT32_TYPE is INTEGER; for INT32_TYPE'size use 32;	32
C	int	32
Ada	FLOAT	32
C	float	32
Ada	SYSTEM.ADDRESS	32
C	(type *)	32

From a software engineering standpoint, only safe types should cross language boundaries. The other data types are not safe to cross the language boundary. Therefore, software programmers in the multilingual software environment must be careful in the following cases:

• It is not safe to directly map a structure or union in C/C++ to the records in Ada, since memory alignment and filling is a function of the compiler chosen. Actually, to interface a C union type in Ada is tricky [Gart95]. Gart presented a method to solve this problem in [Gart95].

• The treatment of enumeration types is not assumed consistent between C and Ada. Many C compilers use discrete numeric values which are zero-offset to implement enumeration types. To assure the enumeration types of two languages are compatible, Ada programmers can use representation clauses to order the compiler to use specific internal

representations for enumeration types. For example, C++ programmers can specify the values of the enumeration constants to define seasons in this way:

```
enum seasons { Spring = 2, Summer = 5, Fall = 8, Winter = 11};
```

In Ada, programmers need to use low-level representation clauses to assure the `seasons` has the same memory representation:

```
type seasons is (Spring, Summer, Fall, Winter);
for seasons use (Spring =>2, Summer =>5, Fall =>8,
                 Winter =>11);
for seasons'size use 32;
```

Software engineers must be careful to handle the situation when a program uses the calculation result of enumeration constants to do something important. Therefore, it is better not to cross the language boundary for enumeration types.

- The fixed-point type in Ada has no analog in C [Hughes91], nor in C++. Programmers may assume an underlying type FLOAT, but the issue of *safe number* and *model number* in Ada is difficult to handle and becomes dangerous if precision of more than a few significant decimal places is concerned. A similar problem occurs when double precision values are required. Not all Ada compilers support double precision floating point values the way most C compilers support it. Therefore, Hughes suggests: "The calculation of extremely precise values should be confined to the stronger of the two languages, with values of only limited precision crossing the language boundary" [Hughes91].

• In Ada, bit fields are handled in *low-level programming* by using representation clauses. Moreover, bit manipulators are also manipulated via Ada's low-level features; unlike C/C++ which provides "bitwise operators" to support programmers for bit manipulation. From the software engineering view point, it is best not to cross language boundaries for bit field types and bit manipulation [Hughes91].

• Strings in Ada and C are not compatible. "In C, any null-terminated sequence of consecutive objects of a single type is considered a string. ... In Ada, strings are either bounded or unbounded" [Hughes91]. They are not compatible in C and Ada. Hughes also proposed a solution for passing an Ada string to C: "The best solution is to ensure that Ada strings are null-terminated by including an ASCII.NUL as the last character of a bounded string and passing a reference to the string using the qualifier "**address**" [Hughes91]. However, the reverse, that of passing a string defined in C to Ada is more problematic.

3.1.5 Constraint Checking

When programmers use the `INTERFACE` pragma, the strict integrity of Ada's strong typing has been infringed. Programmers should not expect that the call interface defined via pragma `INTERFACE` to C can apply the same constraints which have meaning only within the closed world of Ada. That means the constraint exception may not be generated in C code [Hughes91]. Therefore, the Ada data types exported across the C/C++ interface are no longer strongly typed because the Ada runtime system may not be able to detect constraint errors of the cross-boundary data types. "The best rule is not to rely on Ada compiler-generated constraint checking when interfacing to C" [Hughes91]. Hence,

when an Ada program interfaces a C subprogram, software developers must understand they are losing the strong typing feature of Ada.

Although the `INTERFACE` pragma has been effectively replaced by three pragmas `Import`, `Export`, and `Convention` in Ada95, some Ada95 compilers may still support it for upward compatibility [Ada95a]. Software engineers still need to pay attention to this issue in the multilingual software environment.

3.1.6 Pointers and Access Types

"C pointers and Ada access types are not equivalent. ... C pointers must be mapped to objects of type `SYSTEM.ADDRESS` in Ada" [Hughes91]. Moreover, the use of pointer arithmetic is frequent in C/C++ but, in Ada, it is almost impossible without resorting to machine code insertions or unchecked programming [Hughes91]. This implies that it is better to minimize and collect the necessary pointer manipulation into a few software modules, and then place these software modules on the C/C++ side.

3.1.7 Program Wrappers

"Program wrapper" refers to which language will be used to write the main program. There are two choice in this issue: C/C++ or Ada. The first choice cannot use Ada's more valuable capabilities: exception handling, elaboration and instantiation [KSC93]. From the software engineering standpoint, it is simpler to use Ada as the main program [Hughes91, KSC93]. Section 3.2.4 will explain the advantages of the second choice.

3.1.8 Input/Output

In C/C++ and Ada, I/O is not part of the language definition. So, these three languages rely on standard units which are hardware dependent. "Very little exists in common between languages relative to input and output" [Hughes91]. C language assumes there are three standard stream files: *stdin*, *stdout*, and *stder*. The C++ stream I/O ties these three files to *cin*, *cout*, and *cerr*, respectively [Pohl93]. Similarly, Ada has *standard input file*, *standard output file*, and *standard error file*. If the operating system does not support the *standard error file*, the *standard error file* may be associated with the same file as the *standard output file* [Cohen96]. However, there may not be a one-to-one correspondance between these standard I/O files.

"C normalizes all input and output as file I/O, relying on either a file descriptor or on a FILE structure" [Hughes91]. Ada performs file I/O by interfacing with objects of type FILE_TYPE which is a limited private type. The mapping from file descriptor onto FILE_TYPE is tricky. Hughes proposed one trick to circumvent this problem: "One such trick is the evaluation of the limited private objects in a procedure call argument list" [Hughes91].

Mapping of C FILE structures is not a feasible solution because it is not a safe type. Moreover, Ada is a real-time language, but C/C++ is not. In many cases, programmers need to check if the files exist or not and then do some operations on these files. In Ada, if there was a File_Exist function to test whether a file exists or not, this function would only indicate the status of the file when that function is called. In a real-time system, the file may

have gone away after the function was executed. Therefore, in a real-time system, the solution to this problem is to use exception handlers [Uhde95b].

3.1.9 Special UNIX-Related Issues

Since C/C++ does not directly support multitasking capability, programmers on UNIX system may use some special UNIX system calls. When a software application is mixed with C/C++ and Ada, multilingual software developers need to take care to ensure that UNIX system calls are compatible with Ada's tasking capability.

Context switching in UNIX is explicit by using setjump/longjump. However, Ada context switching is implicit in the task model. These two models may conflict with each other such that task thrashing or starvation occurs. "If the degree of explicit control is high, it is advisable to transform the UNIX model into the Ada tasking model" [Hughes91].

Under UNIX, software systems usually use shared memory as an interprocess communication mechanism. Ada83 did not provide the shared memory mechanism for interprocess communication. Ada95 provides protected types where the idea is similar to the shared memory. However, it is not guaranteed that the UNIX shared memory mechanism is compatible with Ada's protected types. It is not too difficult to access the shared memory for Ada code. "However, this is not good software engineering for two reasons: one, it bypasses the protection mechanism implemented in true shared memory systems, and two, it reduces to the data structure encapsulation issue discussed in the section Data Representation and Interoperability" [Hughes91].

3.1.10 Linker Issues

For multilingual software development, the object format generated by one language compiler may not be completely compatible with another language's linker. In this case, multilingual developers may need to convert the record formats and the definition of symbols and string images prior to linking. It may not be difficult but it is an error-prone process [Hughes91].

3.1.11 Tasking

Ada tasks may coexist with C code. Multilingual programmers may be able to make Ada's entry calls visible via pragma `EXPORT` such that C code can call Ada tasks. In the reverse case, Ada task can suitably define a call implemented in C and make it callable within Ada tasks via pragma `INTERFACE` [Hughes91] or the new pragma `Import`.

3.1.12 Pragma Interface, Pragma External Specification.

The use of some pragmas, such as `INTERFACE`, `IMPORT`, and `EXPORT`, may vary from vendor to vendor. Multilingual developers need to carefully study the reference menu provided by the compiler vendor [Hughes91].

3.2 Language Conversion Methods

This section presents some conversion methods from a non-Ada language to Ada. In [KSC93], it presented some approaches to convert non-Ada software systems to Ada: (1) complete redesign and rewrite, (2) incremental functionally equivalent replacement, (3) incremental redesign and rewrite, (4) incremental multilingual rewrite, and (5) automatic translation.

3.2.1 Complete Redesign and Rewrite

To exploit all advantages of Ada, the most thorough method of converting a system to Ada is to do a complete redesign and rewriting. This approach involves reanalyzing, redesigning, implementing and testing of the system and tries to make best use of Ada's software engineering capabilities [KSC93].

If the original software system was badly designed or implemented beyond the original good design, this approach may be the best approach since it may be too difficult to maintain the original code. Simply translating the original code into Ada might result in code no better than the original. Therefore, complete redesign and rewriting of the original software system should provide a system with consequent improvements in maintainability [KSC93].

3.2.2 Incremental Functionally Equivalent Replacement

Incremental functionally equivalent replacement conversion methods can be used when the original design is still valid and the source code was well developed in software engineering principles [KSC93]. This approach replaces a software module with an Ada subprogram (procedure or function but not package) such that the Ada subprogram can function as well as the replaced software module.

This approach has its advantages. One is that it can be performed while the system is still being used [KSC93]. Another advantage is that it is easier to locate problems caused by the conversion when they occur. When the new converted software system does not function as well as before, software engineers can be relatively certain that the problems are in the new converted Ada subprogram.

However, this approach also has its disadvantages. The biggest disadvantage is that the converted Ada subprogram needs to take into account the interface to the other unconverted calling software modules. Since two different language software modules exist, all problems presented in multilingual software engineering in Section 3.1 must be considered.

Another disadvantage is that this approach makes little use of Ada software engineering capabilities. This approach converts only a small software module into an Ada subprogram at a time so no structuring mechanism above the subprogram level is used. Consequently, this approach loses the advantages of Ada's package capabilities. Ada's package capabilities helps software engineers implement several software engineering principles: modularity, data abstraction, and information hiding capabilities [SPC95].

Another issue dealing with the incremental conversion method is the choice of the approach to convert the software system: top-down or bottom-up. Neither way can dodge the interface problems. To decide which direction should be taken in the conversion process, software engineers need to identify the differences and limitations between interfacing the target language from source language and interfacing the source language from target language. Then software engineers should predict the ripple-effects of both approaches in the language conversion process. Finally, the less ripple effect inducing interfacing approach should be chosen. Incremental conversion using both approaches (top-down and bottom-up) is not recommended because interfacing two languages in both directions makes incremental conversion more difficult. If interfacing the source language from the target language is chosen, then top-down fashion should be proceeded in

the incremental conversion. Otherwise, bottom-up will be the better approach to do the increment conversion.

3.2.3 Incremental Redesign and Rewrite

The "Incremental Redesign and Rewrite" approach is similar to the "Incremental Functionally Equivalent Replacement", except an entire subsystem is rewritten rather than just a single module or subprogram. The converted subsystem should be redesigned to make full use of the software engineering capabilities of Ada, including the package concept [KSC93]. This is one advantage over the "Incremental Functionally Equivalent Replacement" approach.

However, the multilingual software engineering difficulties in "Incremental Functionally Equivalent Replacement" still exists in this approach. Software engineers still need to carefully deal with the interface problems between the converted Ada subsystems and the original non-Ada subsystems.

3.2.4 Incremental Multilingual Rewrite

"The result of an incremental multilingual rewrite is a system that is only partially in Ada" [KSC93]. Sometimes, to convert an entire software system into Ada is not feasible. For example, many applications are developed on the top of some libraries or lower-level software components in software architecture. These libraries and lower-level software components may be developed by outside vendors. The software engineers may not be able to convert these libraries and software components into Ada. Also, completely converting the whole software system into Ada may not be a good idea. These lower level

and reusable software components have been tested and used for a long time in the software development system. These software components are reliable. Converting them into Ada code is not very valuable. In these cases, multilingual rewriting is a feasible alternative.

The first decision in this approach is to decide which run-time system will be in control, i.e. which language should be used in the "main" program. There are two options in this issue: the original non-Ada language run-time system or Ada run-time system. The first choice cannot use Ada's more valuable capabilities such as exception handling, elaboration and generic instantiation [KSC93]

Although the original software system may not have any exceptions during the conversion process software engineers may need to include new Ada libraries in which exceptions are already programmed. Usually this kind of trouble is not easy to solve because non-Ada run time systems generally cannot handle exceptions [KSC93]. Even though some run-time systems may handle exceptions, for instance C++, the resolution of the exceptions is different. In C++, unlike Ada, there is no exception type. All exceptions, in C++, are `const char*` type [Johnston]. Therefore, the exceptions generated from Ada libraries may not be handled by C++.

The loss of generic instantiation will inconvenience some software developers. Ada's strong typing property supports the requirement for reliability. However, without generic instantiation capability, Ada cannot easily create families of data structures [KSC93]. Ada's generic capability makes it possible to solve a set of similar but not identical problems with a single program unit [Cohen96]. Without this capability, it is too difficult to write general-purpose, reusable software components.

As compared with exception and generic instantiation problem, the data elaboration problem is easy to solve. Software engineers can implement standard initialization code at the start of each non-Ada module to replace the data elaboration capability of Ada [KSC93].

Since the first choice loses many of Ada's valuable capabilities, the second choice, the Ada run-time system, seems the better plan by far. In this plan, software engineers start on creating a dummy Ada main program that calls the non-Ada main program. The dummy Ada main program only needs to accept the command arguments and then pass them to the non-Ada main program.

Of course, multilingual software engineering issues also need to be addressed in this approach. Software engineers need to decide which parts of the system should remain and which parts of the system should be converted into Ada. Additionally, the conversion sequence of the software components needs to be determinated. A good conversion sequence will ease the language conversion process. As presented in 3.2.2, either top-down or bottom-up approach should be taken. Then the detailed conversion sequences may be determined according to the characteristics of the software application or software engineering criteria. For example, considering reuse, some modules used as public utilities may need to be converted early.

3.2.5 Automatic Translation

The automatic translation approach uses automatic or semi-automatic tools to directly translate the original non-Ada software application into Ada. The quality of the conversion depends on how "smart" the automatic tool is. Generally these tools convert the

original software application into an Ada software application that has the same function as the original software application. However, these tools usually only use those Ada capabilities which were already available in the original language model [Horton 85]. Simply using the automatic translation approach to convert the original language into Ada code may not add any value.

3.3 Object-Oriented Programming in C++ and Ada95

This section introduces how object-oriented features are implemented in Ada95 and C++. By comparing the differences of these two programming languages in object-oriented features, software engineers can find some guidelines in how to use their advantages when facing multilingual software engineering and language conversion issues. The material of this section is obtained from [Martin95], [Balfour2], [Balfour3], [Taft94], [Deitel94], [Jorgens93], and [Cohen96].

3.3.1 The Implementation of Classes

In Ada95, classes are implemented as tagged types. Methods of a class are expressed as subprograms declared within the same package as the tagged types and having the tagged type as a parameter or the result. Attributes of the class are declared as components of the tagged type record.

C++ does not have the concept of a package. A class is declared by using keyword **class**. Then all methods are declared within the class declaration. All methods declared locally in the class have an implicit parameter of the class type [Jergens93]. The attributes of the class are declared within the scope of the class declaration. Table 2 shows the

implementation of a class in C++ and Ada95. Note that the C++ methods, `init_sim` and `pre_draw`, of the `simulation` class, do not have a class as a parameter. These are different with the Ada95 methods.

If programmers want to keep an object constant when calling a method in C++, they need to use the keyword **const**, unlike Ada where this is default. For example, in Table 2, the `pre_draw` method of the `Simulation` class will not change the state of the object if it is called.

Table 2. The implementation of classes in Ada95 and C++ (Adapted from [Jorgens93].)

Ada95	C++
package simulation_class is type simulation is **tagged** with record -- list attributes here time_of_creation : calendar.time; message : text; end record; -- common data member Total_Simulation_No : Integer; -- declare methods below procedure init_sim(sim : in out simulation); -- The default mode of sim is "In", so the -- contents of the object passed in won't -- be changed procedure pre_draw(sim : simulation); end simulation_class;	class simulation { public: // list attributes below calendar::time time_of_creation; text message; // declare methods virtual void init_sim(); // use const to declare a method that won't //change the objects' contents virtual void pre_draw() **const**; // common data member **static** int Total_Simulation_No; };

To allow all objects of a class to share common data in Ada95, programmers can simply declare a variable in the same package where the class was declared as tagged type, like `Total_Simulation_No` in Table 2. For the same reason, C++ programmers need to declare such common data as **static**.

To implement an abstract class in Ada95, programmers can use the keyword **abstract** to create a tagged null record and then declare the abstract class's methods as abstract. In C++, "A class is made abstract by declaring one or more its virtual functions to be pure. A pure virtual function is one with an *initializer of* **= 0** in its declaration" [Deitel94], for instance:

```
virtual void init_sim() = 0;
```

Table 3 shows an example how to implement an abstract class in Ada95 and C++.

Table 3. The implementation of abstract classes in Ada95 and C++ (Adapted from [Jorgens93].)

Ada95	C++
`package simulation_class is` `  type simulation is abstract tagged null record;` `  procedure pre_draw( sim : in out simulation) is abstract;` `  procedure init_sim( sim : in out simulation) is abstract;` `end simulation_class;`	`class simulation {` `public:` `virtual void pre_draw(){};` `virtual void init_sim() = 0;` `};`

3.3.2 Object Creation and Destruction

In C++, *constructors* are used to initialize the objects of a class when the objects are declared, and the *destructors* are used to clean-up the objects of a class when the objects are

destroyed. The constructor of a class is declared as a function that has the same name as the class without a return result. The destructor of a class is also declared as a function without a return result and has the same name as the class with an added "~" prefix. A constructor is implicitly invoked when its associated class type is used in a definition or when call-by-value is used to pass a value to a function [Deitel94]. A destructor is implicitly invoked whenever an object of its class must be destroyed, typically upon block exit or function exit [Deitel94].

Ada95 has similar features called automatic initialization and finalization but these are built on one specific package, called Ada.Finalization. Only the types derived from two predefined types, `CONTROLLED` or `LIMITED_CONTROLLED`, have these capabilities. `Ada.Finalization` declares an abstract tagged type `CONTROLLED` along with three operations:

```
procedure Initialize (Object: in out Controlled);
procedure Adjust     (Object: in out Controlled);
procedure Finalize   (Object: in out Controlled);
```

The default action of these three operations in the Ada.Finialization package does nothing but simply return. It is expected that programmers will override these three operations with their own required operations to more efficiently control the objects. When an object of the derived `CONTROLLED` type, called a *controlled* object is created without an explicit initial value, the `Initialize` procedure will be automatically called. Whenever the controlled object has been assigned, the `Adjust` procedure will be automatically called right after the assignment. The `Finalize` procedure will be automatically called right before the controlled object goes out of existence [Cohen96].

The derived types from LIMITED_CONTROLLED have the same effects except they do not have the Adjust procedure. Since these three operations, Initialize, Adjust, and Finalize, are not provided to be called by the other classes, it is better to declare these three operations in the private section, as shown in Table 4.

Table 4. The implementation of constructor/destructor in C++ and Ada95's automatic initialization and finalization (Adapted from [Jorgens93].)

Ada95	C++
with Ada.Finalization; package simulation_class is type simulation is **new** Ada.Finalization.Controlled **with private**; Total_Simulation_No : Integer; procedure init_sim(sim : in out simulation); procedure pre_draw(sim : in out simulation); **private** type simulation is **new** Ada.Finalization.Controlled **with** record time_of_creation : calendar.time; message : text; end record; procedure Initialize (Object: in out simulation); procedure Adjust (Object: in out simulation); procedure Finalize (Object: in out simulation); end simulation_class;	class simulation { public: void init_sim(); void pre_draw(); simulation(); ~simulation(); static int Total_Simulation_No; private: calendar::time time_of_creation; text message; };

There are two disadvantages of Ada95 in this area [Jorgens93]:

1. Only the derived types from CONTROLLED or LIMITED_CONTROLLED have the initialization and finalization features. Because Ada does not directly support multiple

inheritance, these features are more difficult to use if programmers want to derive them from the other parent types.

2. Only one constructor can be declared and it does not allow the other parameters except the object itself. That prevents Ada programmers from writing a parameterized constructor.

The solution to compensate for these two drawbacks is to create three similar operations corresponding to `Initialize`, `Adjust`, and `Finalize` and explicitly call them at appropriate time. Although this approach loses one of the true value of these features, automation, it still retains control of the objects.

3.3.3 Inheritance

The inheritance feature is implemented by deriving from the parent type. In C++, there are three inheritance types: public, protected, and private. However, "Protected inheritance and private inheritance are rare and each should be used only with great care" [Deitel94]. Therefore, this thesis work only discusses public inheritance and its counterpart in Ada95. In Ada95, the keyword **new** is used to derive a new child class. In C++, the keyword **public** is used to implement public inheritance from the parent class(es) as shown in Table 5. To override a method in the parent class, both languages redeclare the method and reimplement it. When adding a new attribute, Ada95 programmers can add the new attributes in a new record and the C++ programmers can declare new attributes in the new class declaration scope, as shown in Table 5.

To ensure that the actual operation is always alive when a call needs to be dynamically bound, Ada95 only allows a tagged type to be derived at the same scope level as the parent type. It is illegal to derive a new tagged type within a subprogram. For the same problem, C++ has the strange rule: derivations within inner scopes are allowed but their life-time is the same as the outermost scope level [Jorgens93].

Table 5. The implementation of inheritance in Ada95 and C++ (Adapted from [jorgens93].)

Ada95	C++
with simulation_class; use simulation_class; package RDT_Application is type RDT_App is **new** simulation with -- one new attribute record Netobj : RDTNetManager; end record; Total_Simulation_No : Integer; -- override the original methods, init_sim -- and pre_draw, of the parent procedure init_sim(sim: in out simulation); procedure pre_draw(sim: in out simulation); -- Add a new method procedure post_draw(sim : in out simulation); end RDT_Application;	#include "simulation.h" class RDT_App : **public** Simulation { public: // one new attribute RDTNetManager Netobj; // override the original methods, init_sim // and pre_draw, of the parent void init_sim(); void pre_draw(void); // Add a new method void post_draw(); };

The most important difference between Ada95 and C++ in this area is that C++ directly supports multiple inheritance but Ada95 does not. In C++, programmers can derive a class from more than one base type. However, in Ada95, it only allows single inheritance. Ada95 offers other ways to obtain the same effect, but that is not as elegant as the multiple inheritance C++ provides. (This is only one particular type of multiple

inheritance. See [Ada95a] for other cases.) Table 6 shows the way to implement multiple inheritance in C++ and Ada95 for this type of multiple inheritance.

In C++, as shown in Table 6, `window` is a root class which means a simple window. Then `window_with_label` and `window_with_menu` are two classes derived from `window`. To declare a class which has label and menu, duplicated public inheritances from `window_with_label` and `window_with_menu` are used to achieve the multiple inheritance.

In Ada95, to achieve the same result, it is not so simple as C++. First, the root class is declared as a tagged type. Then, two generic packages, `label_mixin` and `menu_mixin`, are created so that two types, `window_with_label` and `window_with_menu` can be specified in `add_label package` and `add_menu package` respectively. Finally, in `add_label_to_menu` package, `label_mixin` generic package is recalled again to specify `window_with_label_and_menu`. Obviously, implementing multiple inheritances in Ada95 is trickier than in C++.

Table 6. The multiple inheritance of a C++ class and the similarity in Ada95 (Adapted from [Jorgens93].)

Ada95	C++
package windows is type window is tagged type with record x_position : position; y_position : position; end windows; with windows; use windows; generic type some_window is new window; package label_mixin is type window_with_label is new some_window with record Label : String; end record; end label_mixin; with windows; use windows; generic type some_window is new window; package menu_mixin is type window_with_menu is new some_window with record menu : menu type; end record; end menu_mixin; with label_mixin, menu_mixin; use label-mixin, menu_mixin; package add_label is new label_mixin(window); type window_with_label is new add_label.window_with_label with null; package add_menu is new menu_mixin(window); type window_with_menu is new add_menu.window_with_menu with null; package add_label_to_menu is new label_mixin(window_with_menu); type window_with_label_and_menu is new add_label.window_with_menu with	class window { position x_position; position y_position; }; class window_with_label: public window { String Label; }; class window_with_menu: public window { menu_type menu; }; class window_with_label_and_menu: public window_with_label, public window_with_menu { ..... };

3.3.4 Polymorphism

"*Polymorphism* is the definition of operations that apply to more than one type" [Cohen96]. In Ada95, polymorphism is achieved by use of tagged types and the classwide type. If using the example in Table 5, it is possible to write a procedure, called init_any_sim, as followed:

```
procedure init_any_sim( sim: in out simulation'class ) is
begin
   init_sim(sim);
end init_any_sim;
```

Here the `simulation'class` is a classwide type. "For each tagged derivation class, there is an implicitly declared type called a *classwide* type" [Cohen96]. For example, if T is a tagged type, then T'Class is T's classwide type which comprises the union of all the types in the tree of derived types rooted at T [APR93]. `Simulation'class` means all types derived from the tagged type simulation including the simulation itself. In `init_any_sim`, it shows that programmers can actually ask the program to execute the specific `init_sim` by providing an actual parameter of the class-wide type `simulation'class`. The Ada run time system will determine which `init_sim` (the `init_sim` in simulation package or the `init_sim` in package RDT_Application) should be executed in run-time.

In C++, the same feature is achieved by declaring the methods as **virtual** in the parent class, like in Table 2. Then C++ programmers can write a procedure as follows:

```
void init_any_sim( simulation* sim )
   sim.init_sim;
};
```

However, the C++ language has at least two drawbacks in this area:

1. In C++, polymorphism is achieved for pointers, not for objects themselves. That seems to be a subtle and error-prone distinction [Jorgens93].

2. In C++, polymorphism for an operation must be stated in the root class. Assume that there are three classes A, B, and C and their inheritance relationship is shown in Figure 3. If programmers later find out it is necessary to override the Method_A1 of the class A in the class C, they need go back and modify the root class A to declare the Method_A1 as **virtual**. In Ada95, programmers do not need to change any parent types [Jorgens93]. From the software engineers' view point, Ada is better than C++ in this situation because it reduces the chance of recompiling.

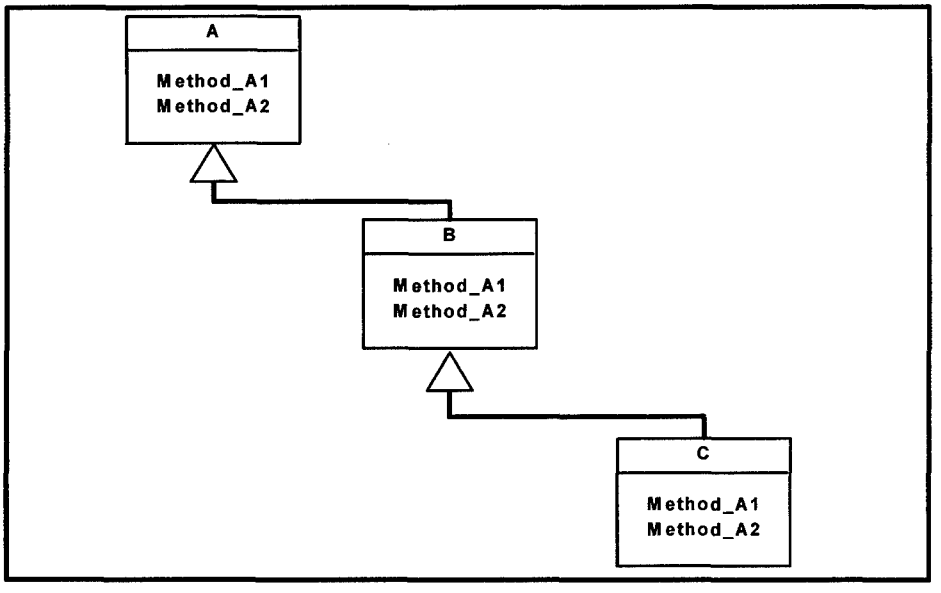


Figure 3. An example to describe the difference of polymorphism between C++ and Ada95.

3.3.5 Visibility

Visibility is the ability that allows a class's attributes and methods to be accessed, used, or read by the other classes. C++ supports three levels of visibility to a class: private, protected, and public. The private attributes and methods of a class are only visible in the class itself. The protected attributes and methods are visible in the class itself and all its child classes, but nowhere else. The public attributes and methods are visible everywhere the class can be viewed [Jorgens93].

Whereas visibility in C++ is decided solely by the class itself, the visibility of attributes and methods of an Ada tagged type is decided by both the tagged type declaration and how it is placed [Jorgens93]. Ada95 supports only the concept of private and public visibility. Ada95 does not support the concept of protected attributes and methods. However, the same idea can be achieved by using Ada95's *child unit*. Child units are packages, subprograms, or generic units locally declared within a surrounding package declaration, but physically presented to the compiler as separate compilation units [Cohen96]. In Ada95, the public attributes and methods are visible everywhere the class is visible, like C++. The private attributes and methods are visible only in the class itself and to its child units. Table 7, an example from [Jorgens93], shows how the Ada95's child package can achieve the same effect as the protected attributes and methods in C++. Also, some guidelines for converting the C++ visibility into Ada's visibility schema are presented in Section 4.5.4.

Table 7. The protected attributes and methods in C++ and the child package in Ada95(Adapted from [Jorgens93].)

Ada95	C++
package alert_system is type alert is tagged private; procedure handle(Object : in out alert); private type alert is tagged record time_of_arrival : calendar.time; message : text; end record; end alert_system; package alert_system.medium_alert_system is type medium_alert is new alert with record action_officer : person; end record; end medium_alert_system;	class alert { public: void handle(); protected: calendar::time time_of_arrival; text message; }; class medium_alert: public alert { public: void handle(); protected: person action_officer; };

3.4 GNAT

3.4.1 The Advantages of GNAT in Mixed Language Programming

"GNU (a self-referential acronym for 'GNU is Not Unix') is a Unix-compatible operating system" [Schonb94]. The GNU C compiler (GCC) is the compiler system of the GNU environment and is the center-piece of the GNU [Schonb94]. "GCC is a retargetable and rehostable compiler system, with multiple front-ends and a large number of hardware targets. Originally designed as a compiler for C, it now includes front-ends for C++, Modula-3, Fortran, Objective-C, and most recently Ada" [Schonb94]. GNAT (an acronym for GNU New York University Ada Translator) is an Ada95 compiler. It couples an Ada95 front end with GCC [Kenner94]. Since GNAT is integrated into GCC, many multilingual software engineering issues can be eliminated by the integration of GNAT and GCC.

Software engineers involved in mixed-language projects can focus on the high level software system instead of the cross-language issues. More information about the GNAT compilation model and the integration between GNAT and GCC can be found in [Schonb94], [Comar94], and [Kenner94].

Table 8. The parallel data types between Ada and C++ in GNAT [SGI]

Ada type	**C type**
Integer	int
Short_Integer	short
Short_Short_Integer	signed char
Long_Integer	long
Long_Long_Integer	long long (GNU C only)
Short_Float	float
Float	float
Long_Float	double
Long_Long_Float	This is the longest floating-point type supported by the hardware. For the MIPS, this is the same as Long_Float, i.e. as the C type double, which is also available as long double in GNU C.
Ada arrays	C arrays
Ada records	C structures
Ada access types	C pointers (except for the case of pointers to unconstrained types in Ada, which have no direct C equivalent.)
Ada enumeration types	C enumeration types (when pragma Convention C is specified)

GNAT provides two ways to interface C and C++ native data types. The first way is to use the types in the package `Interface.C` and the second way is to use standard Ada types [SGI]. The correspondence between Ada types and C types is listed in Table 8 [SGI]. The second way is less portable to other compilers, but will work on all GNAT compilers. GNAT will guarantee the correspondence between Ada types and C types [SGI].

Since most Ada types have corresponding types in C, the data representation interoperability issue (presented in 3.1.3) and safe types issues (discussed in 3.1.4) have been solved by the GNAT compiler. Programmers can directly interface C types in Ada

programs or reversely. Because GNAT has the same center-piece (GCC) as the other front-ends in GNU, calling sequence, data structure layout etc. are automatically compatible [Dewar94]. Therefore, the other multilingual software engineering issues discussed in 3.1, such as call stack ordering (3.1.1), compiler dependencies (3.1.2), and linker issues (3.1.10) are solved by GNAT.

Also, the GCC's exception handling mechanism is intended to be usable by all GCC languages that have exceptions: Ada, C++, and Modula-3 [Schonb94]. That allows mixed-language programs to function in the presence of language-specific exceptions and exception handlers.

3.4.2 C++ Low-Level Interface Capability

Interfacing C++ classes from Ada95 code was a critical issue in this thesis work. There were two reasons why it was necessary to interface C++ in this thesis work: (1) Incremental conversion was used: In the incremental conversion process, selected C++ code was translated into Ada95. Then, the new Ada95 code and the existing C++ code were combined to test the converted result. Therefore, it was necessary to interface C++ code so that the new Ada95 code could work together with the remaining C++ code.

(2) The foundations of RDT, ObjectSim and Object Manager, were implemented in C++: Since these foundations would not be converted to Ada95, the code that was converted to Ada needed to interface to the existing C/C++ software foundations.

Because of the above two reasons, it was necessary to interface C++ from Ada95 for this thesis work. Fortunately, GNAT provides some C++ *low-level* interface capabilities.

When converting C++ into Ada95, the most important C++ interface capabilities are to interface C++ classes, including their attributes and methods, such that Ada95 programmers can still use the object-oriented concept to reuse the existing C++ classes and develop their software applications. GNAT provides some non-standard pragmas to interface C++ classes. These pragmas are summarized in Table 9 and the detail description are presented in [Comar].

Table 9. The pragmas for interfacing C++ classes to Ada95 (Adapted from [Comar, SGI].)

Pragma Name	Profile	Comment
CPP_Class	pragma CPP_Class(Entity => Type);	It indicates that Type corresponds to a declared C++ class type
CPP_Vtable	pragma CPP_Vtable(Entity => Type, Vtable_Ptr => Field_Name, Entry_Count=>Static_Number);	Entry_Count is the number of virtual functions on the C++ side. This pragma asks the compiler to allow dynamic dispatching through the vtable pointer Field_Name.
Import	pragma Import (Conversion => CPP, Entity => Subprogram/Variable, External_Name => Whatever, Link_Name => "manually mangled_name");	This pragma tells the compiler that the subprogram or variable name specified by the Entity parameter is implemented in C++ and the subprogram or variable can be linked with Ada programs through the Link_Name parameter.
CPP_Virtual	pragma CPP_Virtual(Entity => Subprogram, Vtable_Ptr => Field_Name, Position => Static_Number);	This pragma provides necessary information for dispatching. The Position parameter tells the compiler the position of the subprogram specified by the Entity parameter in the Vtable,.
CPP_Constructor	pragma CPP_Constructor(Entity => Fname, Link_Name => "manually mangle name");	This pragma specifies a C++ constructor, called Fname, and this constructor can be linked with Ada program by the Link_Name..
CPP_Destructor	pragma CPP_Destructor(Entity => Pname);	This pragma specifies a C++ destructor, named Pname.

Basically, a C++ class with any virtual functions can be equally viewed as an Ada tagged type. That means mixed programmers can interface a C++ class that has any virtual functions by declaring a corresponding tagged type on the Ada side and use the above pragmas to interface the C++ class as an Ada tagged type. However, a C++ class without

any virtual function may not be fully compatibly interfaced as an Ada tagged type. This is because they may have different memory layouts from the compilers' view point. A C++ class without any virtual function can be interfaced as a regular Ada record type. However, using this fashion to interface a C++ class without any virtual function is losing its reusability because a regular Ada record type can not be derived any further. The inheritance mechanism of OOP cannot be used any more. Hence, it is better is to change the original C++ classes which have no virtual function into the classes which have a virtual function by adding a dummy virtual function in them. Therefore, the original C++ classes can be interfaced as an Ada tagged type without losing their reusability.

In order to demonstrate the capabilities of the GNAT's C++ low-level interface capability, an example which shows how to derive an Ada tagged type from an existing C++ class, has been extracted from [Comar]. The example, shown in Program 1, starts out with these existing C++ classes: `Origin`, `A`, and `B`. In order to interface from the Ada95 subprogram to the existing C++ classes, `A` and `B`, two package specifications, which use some of GNAT's C++ low-level interface capabilities, must be created. The package specifications of `A` and `B` are shown in Program 2 and Program 3, respectively. Once done, a new Ada tagged type, called `C`, can be derived from the C++ class `B`. The existing methods which belong to C++ class `A` can be overridden, as shown in Program 4. Program 5 is an Ada main program to test the interface package specifications.

```
class Origin {
 public:
  int o_value;
};
class A: public Origin {
 public:
  void non_virtual (void);
  virtual void overridden (void);
  virtual void not_overridden (void);
  A();
  int a_value;
};
class B: public A {
 public:
 virtual void overridden (void);
 B();
 int b_value;
};
```

Figure 4. Program 1: Some existing C++ classes (Adapted from [Comar].)

```
 with Interfaces.CPP;
 use  Interfaces.CPP;
 package A_Class is
 --
 --  Translation of C++ class A
 --
  type A is tagged
     record
       O_Value : Integer;
       A_Value : Integer;
       Vptr    : Interfaces.CPP.Vtable_Ptr;
     end record;

 pragma CPP_Class (Entity => A);
 pragma CPP_Vtable (Entity => A, Vtable_Ptr => Vptr, Entry_Count =>2);
 --   Member Functions
 procedure Non_Virtual (This : in A'class);
 pragma Import (CPP, Non_Virtual, "non_virtual", "_non_virtual_1A");

 procedure Overridden (This : in A);
 pragma CPP_Virtual (Entity => Overridden, Vtable_Ptr => Vptr, Entry_Count => 1);
 pragma Import (CPP, Overridden,"",  "_overriden__1A");

 procedure Not_Overridden (This : in A);
 pragma CPP_Virtual (Not_Overridden);
 pragma Import (CPP, Not_Overridden,"",  "_not_overriden__1A");

 function Constructor return A'class;
 pragma CPP_Constructor (Entity => Constructor);
 pragma Import (CPP, Constructor, "A", "__1A");

end A_Class;
```

Figure 5. Program 2: The interface package specification of the A class (Adapted from [Comar].)

```
with Interfaces.CPP;
use  Interfaces.CPP;
with A_Class;
package B_Class is
--
--  Translation of C++ class B
--
  type B is new A_Class.A with
    record
      B_Value : Integer;
    end record;

pragma CPP_Class (Entity => B);

function Constructor return B'class;
pragma CPP_Constructor (Entity => Constructor);

procedure Overridden (This : in B);
pragma CPP_Virtual (Overridden,Vptr,1);
pragma Import (CPP, Overridden, "Overridden", "_overriden__1A");

end B_Class;
```

Figure 6. Program 3: The interface package specification of the B class (Adapted from [Comar].)

```
with Interfaces.CPP;
use  Interfaces.CPP;
with B_Class;
package Ada_Extension is
  type C is new B_Class.B with
    record
      C_Value : Integer := 3030;
    end record;

-- no more pragma CPP_Class, CPP_Vtable, or
-- CPP_Virtual: this is a regular Ada tagged type

procedure Not_Overridden (This : in C);
pragma CPP_Virtual (Not_Overridden,Vptr,2);
pragma Import (CPP, Not_Overridden,"", "_not__overriden__1A");
end Ada_Extension;

with Text_IO; use Text_IO;

package body Ada_Extension is

    package Int_IO is new Text_IO.Integer_IO(Integer);
      procedure Overridden (This : in C) is
      begin
        Put(" in Ex6_If.Ada_Extension.Overridden, a_value =");
        Int_IO.Put(This.A_Value);
        Put(",b_value = ");
        Int_IO.Put(This.B_Value);
        Put(",c_value = ");
        Int_IO.Put(This.C_Value);
        New_Line;
      end Overridden;
end Ada_Extension;
```

Figure 7. Program 4: A package that derive a new Ada tagged type from the existing C++ classes (Adapted from [Comar].)

```
with A_Class;use A_Class;
with B_Class;use B_Class;
with Ada_Extension;use Ada_Extension;
procedure Ada_main is
  A_Obj : A_Class.A;
  B_Obj : B_Class.B;
  C_Obj : Ada_Extension.C;
  procedure Dispatch (Obj : A_Class.A'class) is
  begin
    Overridden (Obj);
    Not_Overridden (Obj);
  end Dispatch;
begin
  Dispatch (A_Obj);
  Dispatch (B_Obj);
  Dispatch (C_Obj);
end Ada_main;
```

Figure 8. Program 5: An Ada procedure to test the GNAT's C++ low-level interface capability (Adapted from [Comar].)

The main difficulty in using GNAT's C++ low-level interface capability is that it's time-consuming. GNAT's C++ low-level interface capability is defined for third party vendors, not for programmers [Comar]. In order to use this capability, the following must be done.

- The correct manually mangled name for Link_Name parameters must be found by hand. When a program accesses to an object which is defined in another programming language, the accessing programming language linker must know the link symbol name of the accessed object. So, the linker can link all programs together. Otherwise, the foreign object cannot be resolved in the linking phase. The *manually mangled name* of a object is the link symbol name which is named by the foreign language compiler. By providing the correct manually mangled name, the linker can correctly link the link symbol name of the object. Since GNAT is only an Ada compiler, in order to understand the interfaced C++ classes, Ada programmers must tell the GNAT linker to link the interfaced C++ classes from the C++ object files. Manually mangled names form the bridge to connect the GNAT

linker and the interfaced C++ object files such that the GNAT linker can correctly link the existing C++ object files with the Ada code. The C++ compiler can be forced to compile the existing C++ code without involving any linking actions. Then, the **nm** command is used to find the appropriate Link_Name. To find the appropriate Link_Name by hand is not difficult but it is time-consuming. Because this is not a typical activity for traditional single-language programmers, care must be taken so that the correct link name is identified.

- The C++ user-defined types must be converted into Ada. Most complex and large systems define many user-defined types for their system-wide use. Converting native C types into Ada is easy, but converting user-defined types is difficult and time-consuming. Compilers can easily identify user-defined types that may be declared hierarchically in different header files in most cases. However, for a human, it is a difficult and error-prone process to hierarchically search the user-defined types and then make correct type conversions.

- To derive a tagged type from a C++ class, all attributes and methods of the C++ class and its ancestors must be declared. However, manually creating package specifications to interface the existing C++ classes is very difficult because cross-references among the C++ classes which are derived from all its ancestors must be made. Therefore, the interface between the higher level application and low level foundation of the software architecture cannot simply be cut.

For example, consider Figure 2 in Section 2.2. In this conversion case, five package specifications to interface the existing ObjectSim C++ classes must be created: `Simulation`, `Base_Net_Player`, `Attachable_Player`, `Simple_Terrain`,

and `View`. However, in order to create the interface package specification for `Base_Net_Player`, care must be taken to assure the attributes and methods declared in `Player` are declared in the interface package specification of `Base_Net_Player`. Further, this may imply that the data types used by `Player` and `Base_Net_Player` also need to be converted to the corresponding Ada data types.

These tasks can be significantly simplified if done by automatic tools. It is very time-consuming to manually interface C++ classes, especially for large and complex systems. If the existing C++ classes are part of a large inheritance tree and many user-defined types exist, then many package specifications for all directly and indirectly included C++ header files must be created.

It is also possible to interface an Ada tagged type as a C++ class from C++ programs. However, interfacing Ada programs from C++ programs is not recommended for this incremental conversion work. Using GNAT to interface Ada modules from C++ programs is the better way to interface. It is more difficult to go from C++ modules to interface Ada programs. When interfacing Ada modules from C++ programs, in addition to needing to export the External_Names of Ada subprograms or variables, it is also necessary to declare these interfaced subprograms or variables as external in the external declaration section. Modifying all C++ modules which refer to these subprograms or variables is required. Therefore, the ripple effect to interface Ada programs from C++ programs is larger than to interface C++ programs from Ada programs.

To convert C++ programs into Ada programs, it will be best to interface C++ code from the Ada side rather than to interface the Ada programs from the C++ side. GNAT's

C++ low-level interface capabilities also allow to interface an Ada tagged type from C++ programs. However, only interfacing C++ classes from Ada programs is recommended in this thesis work. Appendix I shows an example to interface an Ada tagged type from C++ programs. In that example, it is clear that interfacing an Ada tagged type from C++ programs are worse than interfacing a C++ class from Ada programs. To interface converted Ada modules, some C++ software modules must specify the link symbols of all member functions of the converted Ada tagged types in their external declaration sections, if the C++ software modules call the member functions of the converted Ada tagged types. This causes the ripple effects to be spread over the other unconverted C++ software modules. On the other hand, if only interfacing C++ classes from Ada programs is used, all needed changes will be limited within the converted Ada modules and the original C++ modules will not be changed. Therefore, interfacing an Ada tagged type from C++ programs is not recommended. Because only interfacing C++ classes was used, top-down conversion is the better choice for converting a C++ software application into Ada in this thesis work.

3.5 Adabindgen

Adabindgen, standing for "Ada Binding Generator", is a tool created and supported by SGI. It can read C++ header files and generate the Ada package specification that contains GNAT's pragmas to interface the existing C++ classes. Then, the existing C++ code can be reused without reimplementing the package body. It helps mixed-language programmers focus on high level programming issues instead of the GNAT's C++ low-level interface issues. This tool automatically gives a correct link name for a member function or variable. Hence, the correct manually mangled name need not be searched for.

It also automatically searches all applicable header files to convert all C++ types, including user-defined types, into Ada types. Since all C++ header files can be automatically converted into Ada package specifications, creating Ada package specifications for all the existing C++ classes' header files is possible and mixed language programming is feasible. However, the philosophies behind C++ and Ada are quite different. This makes directly translating C++ header files into Ada package specifications more difficult. Basically, Ada has stronger restrictions and some restrictions result in difficulties in correctly converting C++ header files into Ada package specifications. Therefore, sometimes, Adabindgen may not be able to correctly convert C++ header files into Ada package specifications because of the essential differences between C++ and Ada.

Some of the essential differences between the two languages are listed below:

1. In C++, circular references can exist. For example, a class header file can reference (via the **include** preprocessor directive) a second header file which directly or indirectly references the first header file. This situation is not allowed in Ada. Ada does not allow a package to use the **with** clause to access another package that also uses the **with** clause to directly or indirectly access the original package. This so-called mutual dependency problem will be discussed in the next chapter.

2. In C++, a member function of a class in a C++ header file can be directly implemented. In Ada, a member function cannot be implemented in a package specification.

3. The visibility of a C++ class's attributes and methods has three levels: public, protected, and private. Only public and private visibility is supported in Ada95. Protected attributes and methods must be carefully translated into Ada.

4. C++ is a case-sensitive language but Ada is not. For example, `FONTA` and `FONTa` are different variables in C++, so they have different link names. However, `FONTA` and `FONTa` are the same variable in Ada. Adabindgen can generate two pragmas such that these two variables in C++ can be imported to Ada with two different link names, but the entity names are still the same for the Ada compiler.

Because of these differences, converting C++ programs into Ada programs is not simply a matter of converting a language's data types and statements into another language's data types and statements. To some extent, one language's programming style must be converted into the other language's programming style. The next chapter will present a methodology to accommodate these differences.

4 Methodology

Because of the essential differences between C++ and Ada, as well as the restrictions of GNAT's C++ low level interface capability, directly translating from C++ to Ada, as presented in [Uhde95b], may not be straightforward. In contrast to this approach, this thesis presents a feasible incremental language conversion procedure consisting of the following phases:

- Reorganize the software application
- Break mutual dependencies
- Create interface package specifications
- Convert C++ code into Ada
- Embellish

4.1 Reorganizing the Software Application

Reorganizing the original software application consists of separating specification from implementation for each software module in the original C++ application so that each header file has only one class's declaration and the implementation of the class is in the corresponding source-code file. The products after this phase are still C++ programs. After this phase, the software system should be rebuilt to verify that the reorganized software system has the same functionality as its original code. The goal of this phase is to reorganize the C++ application so that it looks like the package style of Ada programs.

In Ada, the *package* is a principal program structure that supports several software engineering principles: information hiding, abstraction, separation of specification and implementation, and modularization [SPC95]. Two goals of using Ada in large and complex software projects are: division of large programs into manageable modules and generation of reusable software components. Both of these goals are supported in Ada by the *package* [Cohen96].

A C++ header file plays a role similar to that of the package specification in Ada and a C++ source-code file plays the role of the package body in Ada. So, C++ file modules loosely correspond to an Ada package, however Ada's package has several advantages over C++. First, Ada package specifications and bodies can be compiled separately, but C++ header files cannot be compiled separately. The second advantage of Ada is that Ada strictly separates the specification in the package specification from implementation in the package body. From the software engineers' point of view, this is an advantage because it forces programmers to separate software specification and implementation. However, C++ programmers can implement *inline* functions within a header file. A C++ inline member function is a member function whose implementation is contained in a class declaration in a header file. From the software engineers' point of view, this exception allows programmers to program in a sloppy fashion.

In order to make further conversion processes smoother and let the converted application take advantage of the package concept of the Ada language, it is necessary to reorganize the original application so that it has Ada's package style. Actually, this style is also a good programming style in C++.

The necessary knowledge in this phase is only the high-level structure of the software system and the C++ program structure mechanism. Detailed language syntax knowledge and programming skills are unnecessary. In this phase, the following issues must be considered:

1. Global variables should be collected and only declared once in one module. The other modules should declare global variables as **extern**. In order to avoid duplicate declarations of the global variables, it is better to declare all global variables in one module and use preprocessor directives to tell the C++ compiler to correctly include global variables. Here is a good style for the global variable declaration header file:

```
#ifndef _MAIN_PROGRAM_
<normal global variables declaration>
#udef _MAIN_PROGRAM_
#else
<declare the global variables as external>
#endif
```

Then, only the main program should include the global variables header file in this fashion:

```
#define _MAIN_PROGRAM_
#include "global.h"
     :
     :
```

2. Logically in C++, `#include` means to copy the included header files into this module. Therefore, developers need to carefully use preprocessor directives to protect the software module from multiple inclusion and circular references, because multiple inclusion and circular references make the program extremely difficult to debug [Martin95]. A good C++ programming style to prevent this problem is to use the following preprocessor directives in sequences [Martin95]:

```
#ifndef file_name_symbol  // If file_name_symbol has not been defined,
#define file_name_symbol  // then define file_name_symbol.
<the rest of the module>
#endif
```

Then the other header files which need to include this header file should have the following style:

```
#ifndef file_name_symbol  // If file_name symbol has not been defined
#include file_name        // then include file_name.
#endif
      :
      :
```

3. Strictly separating implementations from header files should be done. The implementations of inline member functions should be moved to its source-code file. Even though the programming style of inline functions can improve performance, it is not good software engineering practice [Deitel94]. Hence, programmers should separate C++ class implementations from C++ class declarations. For example, a C++ class constructor can be declared and implemented in the following way:

```
class Class_A {
public:
  Class_A(){< implementation >};
};
```

Instead of declaring a class in this way, programmers should strictly separate declarations from implementations in this way:

```
class Class_A {
public:
  Class_A();
};
```

Then implement the constructor in the source-code file:

```
Class_A::Class_A(void){ < implementation > };
```

There are two major reasons for this programming style. First, if an inline function temporarily contains nothing, C++ compilers may not generate a link symbol for this inline function since the inline function contains no code. Most C++ compilers may optimize the object code and will not give the inline function a link symbol, because they will not be used in the actual program execution. Therefore, mixed language programmers cannot use **nm** to find the "manually mangled name" to interface it. The second reason is that the header file won't need to be modified if the implementation is fulfilled in the future. For applications where performance is critical, the functions can be implemented as Ada inline subprograms during the fifth phase by using `Inline` pragma. Therefore, the subprograms can be expanded in line at the call site [Ada95b].

4. A dummy virtual function should be added into the root classes which have no virtual functions. Since a C++ class without any virtual function does not completely correspond to an Ada tagged type, a dummy virtual function should be added in order to have an absolute one-to-one mapping relation. A C++ class may be converted to a regular Ada record type, but it cannot be used as a root class or be used to derive a new child class. A C++ class without any virtual functions also may be interfaced as an Ada tagged type. However, in this case, the C++ language compiler (SGI CC compiler), and GNAT may use different memory layouts for the objects that should be in the same class from the object-oriented programmer's view point. This is a dangerous interface option, because an Ada access type object may point to the wrong position of an interfaced C++ class array. Therefore, software engineers should convert this C++ class with the dummy virtual function into a fully compatible Ada tagged type in later conversion phases.

5. A member function which modifies the class data attributes and has a return value must be changed into a member function without a return value, because an Ada function cannot have **in out** mode parameter. Appendix J presents the rules for converting C++ functions into Ada subprograms. It also explains why it is necessary to change these kinds of C++ class member functions into the member functions without a return value. These procedures must be done in this phase, so the changed C++ class member functions can be directly converted into Ada procedures in the fourth phase.

In this conversion case, the original RDT main program encapsulated several classes in the main module. After reorganizing the original RDT application, each software module contained only one class declaration or implementation. The software modules of reorganized ObjectSim and RDT are listed in Appendix B. Appendix J lists the changed member functions and the ripple effects which are related to the issues discussed above in the previous paragraph. The reorganized RDT has been rebuilt to do some high-level function tests. Since the detailed operation functions of RDT were not well documented before, no formal testing procedures were followed to verify whether its original functionality has been completely meet. Based on some rough high-level function tests, the original high-level functions of RDT seem to be retained.

4.2 Breaking Mutual Dependencies

Mutual dependency is a circular reference among software modules. In C++, a module may use the `#include` preprocessor directive to refer to some declarations in the other modules. Similarly, in Ada, a compiler unit can use **with** clauses to refer to some declarations in the other packages. There is one important difference between these two

languages in this issue and this difference makes direct conversion from C++ to Ada more difficult.

In C++, programmers can have co-dependent class declarations in different header files. C++ programmers can first roughly declare a C++ class and later declare the detailed contents in another header file. For example, C++ programmers can declare a member function which has a parameter and the type of this parameter is another C++ class:

```
#include "Class_A.h"
class Class_A;
class Class_B {
public:
   << data attribute declarations >>
   void B_func(Class_A* A);
};
```

The same situation may be needed for `Class_A` declaration in file class_A.h to refer to `Class_B`:

```
#include "Class_B.h"
class Class_B;
class Class_A {
public:
   << data attribute declarations >>
   void A_func(Class_B* B);
};
```

In this case, if we simply directly convert C++ header files into Ada package specifications, the results will look like the following Ada package specifications:

```
with class_B;          |          with class_A;
package class_A is     |          package class_B is
:                      |               :
:                      |               :
end class_A;           |          end class_B;
```

This causes mutual dependency problems because the Ada language does not allow programmers to use **with** clauses in a package specification to access another package specification if the referred package specifications directly or indirectly access back to the original package specification. This kind of circular references between `Class_A` and `Class_B` is a mutual dependency.

Obviously, some reengineering work is needed to conquer mutual dependencies in the language conversion process. Two choices can be taken [Quiggle96a]:

1. Modify the existing C++ header files to eliminate mutual dependencies prior to converting the C++ header files into Ada package specifications or
2. Modify the converted Ada package specifications to eliminate the mutual dependencies.

The disadvantage of the first option is that software engineers need to reengineer the original software system. However, some original functions provided by the low-level software architecture will be lost. For instance, in this RDT conversion case, some mutual dependencies exist in the header files of ObjectSim which is the foundation of RDT. If software engineers break these mutual dependencies, some original functions provided by ObjectSim will be lost. Therefore, referring to Figure 1, since RDT is an application on the top of ObjectSim, some extra work must be done to recover the lost functions that were originally provided by ObjectSim.

The second option can preserve all original functions but software engineers can not identify whether they successfully broke mutual dependencies until the main program has been converted into Ada code. On the other hand, the first option can verify the success of breaking mutual dependencies in the early phase. After modifying the existing C++

source code to break the mutual dependencies, the software application can be rebuilt to see whether the software system can function as well as before.

If the first option is chosen, a system dependencies diagram should be drawn to find all mutual dependencies in the software system. The *system dependencies diagram* is a diagram which describes the inclusion dependencies among all software modules. In a system dependencies diagram, each node stands for a C++ header file and a directed link from node A to B means that header file A includes header file B. If there is a cycle between any pair of nodes, the two nodes are mutually dependent on each other. A decision then has to be made as to which directed link in the cycle should be broken to solve the mutual dependency problem.

Appendix C.3 presents a way to minimize the loss of functionality in a software architecture when breaking mutual dependencies if the first option is chosen. Chapter 12 of [Cohen96] presents another method to break mutual dependencies. If the second option was chosen, Cohen's method is another way to solve mutual dependency problem.

This thesis effort to convert RDT used the first option. All mutual dependencies occurred in ObjectSim. The simplified system dependencies diagram of ObjectSim, the modified ObjectSim header files, and the rationale are presented in Appendix C. The modified ObjectSim has been informally verified by rebuilding RDT and interfacing it to a small application called "testsim".

4.3 Creating Interface Package Specifications

In this phase, the necessary interface package specifications to interface the existing C++ code must be created. Usually one C++ header file is converted into one Ada package specification to interface the corresponding C++ source-code file. Adabindgen is a good

tool in this phase. However, because of some technique problems[1], the package specifications generated by Adabindgen must be examined and modified, if necessary, so that the generated package specification can correctly interface the existing C++ code.

Unit testing for each interface package specification is recommended in this phase. It assures each created Ada interface package specification can correctly link to the existing C++ code. This step will help to avoid problems in the later conversion phase. In order to do unit test in this phase, the interface package specifications should be created and tested in a bottom-up fashion according to the system dependencies diagram. Since all mutual dependencies were broken in the previous phase, the system dependencies diagram should now be a diagram without any cycles. If a top-level header file was converted into an Ada package specification, the package specification cannot be compiled until all the original header file's included header files have been converted into Ada package specifications as well. Moreover, if the top-level package specification requires modification and recompilation, then all its referred to package specifications need to be recompiled again as well. Therefore, creating the interface package specifications in a bottom-up fashion is better. After a bottom-level header file has been converted into an Ada package specification and has passed the unit test, it will not be further modified. This reduces the chance of recompiling.

Whether they are used or not, two basic access types must be defined in the interface package specifications. One is `X_Class_Ptr` and another is

---

[1] During this thesis work, adabindgen was still under testing. It was an alpha-version software product. In some cases, adabindgen cannot convert a C++ header file into a perfect Ada package specification.

X_Class_Classwide_Ptr where X is the class name. They should be defined as follows:

```
type X_Class_Ptr is access all X_Class;
type X_Class_Classwide_Ptr is access all X_Class'class;
```

These two basic access types are used very often, especially for converting a polymorphic member function. Declaring them in the early phase can avoid defining them later when it is found necessary to define these two types. The advantage is that the interface package specifications will not need to be compiled again.

Also, the problems raised due to C++ being case-sensitive should be solved in this phase. There is a semi-automated method to solve this problem. One of the collision variable names can be renamed and used as the entity name in the "Import" pragma, there by associating it to the same link name. In this way, the renamed variable in the Ada program can be used while the original variable name is used in the C++ program. For example, FONTA and FONTa are two different variables in C++. Adabindgen will use import pragma to interface these two variables between C++ and Ada:

```
pragma Import (C, FONTA, "FONTA",.....);
pragma Import (C, FONTa, "FONTa",...);
```

However, in Ada, the entity name FONTA and FONTa are same. So, GNAT will think one entity has two different link names and generate a compiler error. The better way to solve this problem is to rename FONTa and use the name that will be used in Ada programs. For instance, the above two Import pragmas should be modified in the following way:

```
pragma Import (C, FONTA, "FONTA",.....);
pragma Import (C, Little_FONTa, "FONTa",...);
```

This way, C++ code of the existing program need not be modified and `Little_FONTa` will be linked to the original C++ link name `FONTa`.

Since the visibility of a C++ class is quite different than the visibility of Ada95's tagged types, converting the visibility of a C++ class into the visibility of an Ada tagged type is not recommended in this phase. It is better to temporarily convert all member functions and attributes of a C++ class into public methods and attributes of an Ada tagged type first and then convert the visibility of a C++ class into Ada's information hiding schema at the last phase (Section 4.5).

There are at least three reasons to support this argument. First, it does not hurt anything for language conversion. Second, it keeps the purpose of this phase simple and achievable. The goal of this phase is to correctly interface the existing C++ code. If the visibility of a C++ class is converted into the visibility of an Ada tagged type in this phase, it will be difficult to identify the actual cause of problems should they occur in this phase. The problems may be caused by incorrect visibility conversion or erroneously interfacing the existing C++ code. Third, it is not impossible to convert a C++ class's visibility into Ada's visibility schema but it is complicated, especially when a C++ class has public, protected and private attributes and member functions mixed together. "Thus, translating from C++ to Ada95, what is needed is not only a mechanism to implement the C++ concept of protected visibility in the Ada95 translation, but also a mechanism to change visibility within an Ada95-derived class" [Uhde95b]. The visibility problem of a C++ class is not a

simple problem of the class itself. It is related to all its derived classes and the other classes existing in the system. Simply focusing on a class's visibility conversion is not adequate. Usually, that only solves the problems for one class. Also, usually the problem is solved based on the philosophy of C++ and not Ada's mechanism at this point. This is because the original C++ programs were designed according to C++ mechanisms. Simply converting the visibility of a C++ class into Ada results in the C++ mechanism being implemented in Ada. The original system design must be modified to use Ada's mechanism instead of simply translating the C++ visibility concept into Ada95. Therefore, it is better to make sure that the converted Ada programs meet the original functions of the original C++ programs in the early phase. Then the visibility of each class is examined. The converted Ada program is then reengineered by using the visibility mechanism and hierarchical library program structure of Ada95 to give appropriate visibility to each class. In summary, converting the visibility of a C++ class in the last phase (Section 4.5) is recommended.

One other point should be made in this phase. To convert a header file that contains global variable declarations, each global variable must be *exported* to the C++ code instead of *importing* global variables into Ada. Otherwise, unresolved link symbol problems will be encountered after the whole system is converted into Ada because the C++ global variables declaration header files are not used anymore.

4.4 Converting C++ Code into Ada Programs

The goal of this phase is to convert the C++ software into Ada. Modules can be converted one at a time. A software module can be a subprogram or a class. Since interfacing an Ada tagged type from a C++ program is not recommended, the C++ main

program must be converted first. Then the other C++ code should be converted into Ada programs in a top-down fashion. The objective in this phase is to convert each statement of the original C++ code into Ada expressions. Therefore, the knowledge to map C++ statements into Ada expressions is required

Most difficulties encountered in this phase deal with interfacing the C++ system libraries. In the SGI operating system, IRIX 5.3, SGI has created Ada interface packages to interface the C++/C system libraries. The `Convention` pragma need not be used to interface the C/C++ library functions. Most of the time, the appropriate Ada specifications for the C/C++ system library functions are available and the problem becomes one of figuring out how to directly make a correct C++/C system library call in Ada. The problems are not difficult but may take time for novices.

If the second option presented in Section 4.2 is chosen to solve the mutual dependency problems, then it should be applied in this phase. Also, after the main program is converted into Ada, the software system should be rebuilt to verify whether breaking mutual dependencies was successful.

After the C++ main program is converted into Ada, a C++ function can be converted into an Ada subprogram in a top-down way. At first, the C++ functions at the top level of the *calling dependency diagram* must be converted first, because interfacing Ada programs from C++ programs is not recommended in this thesis work.

The calling dependency diagram of a software system is a diagram that describes the calling relationship of procedures and functions in the software system. The diagram shows which functions are called by a specified function and which functions call this

specified function. This diagram is a roadmap to guide in picking which functions should be converted in the next step. The calling dependency diagram of RDT is included in Appendix D.

A C++ function can be converted into an Ada subprogram according to the rules presented in Appendix J. One problem may be encountered in this phase. In C++, it is allowed to declare a local **static** variable in a C++ function. In this way, the local **static** variable will retain its value when the function is exited. So, when the function is called next time, the **static** local variable will contain the value it had when the function last exited [Deitel94]. In this respect, Ada has no **static** variables, but the same result can be achieved by declaring the variable in the package level rather than in the function. Using this method, the value of the variable can be preserved if the function is called the next time. It is recommended to declare the variable in the package body. Because the local variable is only used in the function, it is unnecessary to declare it in the package specification to be visible by other packages.

When the implementation of a C++ function is converted into an Ada subprogram in a package body, the `Import` pragma can be commented out in the Ada interface package specification. In this way, GNAT will compile and link the converted Ada subprogram instead of the existing C++ function.

After a C++ function or several related C++ functions are converted into Ada subprograms, the software system should be rebuilt to test if the converted subprograms work as well as the original C++ functions. This completes a single subprogram

conversion cycle, which is continuously repeated until all C++ functions are converted into Ada programs.

After all methods in an interface package are converted into Ada programs, all import pragmas should have been commented out. At this point, the `CPP_Class`, `CPP_Virtual`, `CPP_Vtable` pragmas and the `Vptr` declared in the tagged type record can be removed. Because, at this point, the class has been completely converted into Ada implementation, all of GNAT's C++ low-level interface pragmas in the package specification are no longer needed any more. Also because C++ functions were converted in a top-down way, no other C++ functions will call this converted Ada subprogram.

4.5 Embellish

Even though all C++ code was converted into Ada programs after the previous phase, some embellishing work should be done to promote the quality of the converted Ada program. Possible embellishing works are renaming, polymorphism perfection, recovering subprogram parameter types, use of Inline pragmas, class visibility remapping, and program structure reorganizing.

4.5.1 Renaming

Some renaming work is necessary in this phase. In C++, a class is defined as a type and declared in a header file. Basically, the class name is the same as the header file name. However, in Ada, a class is implemented as a tagged type in a package. Ada does not allow a tagged type to have the same name as the package name. Therefore, an additional appropriate name for the package is necessary. In the third phase of this conversion

methodology, Adabindgen converted a C++ class name into an Ada tagged type name and the header file name was taken as a package name. For example, if a C++ class declaration called Class_A is made in file class_A.h, Class_A will be converted into an Ada tagged type and the package name will be class_A_h, It is recommended to keep this naming convention in the third phase but rename them in the fifth phase, because this is not a good naming convention. Two naming conventions have already been proposed by [Cernosek93] and [Rosen95]. Both conventions take the class name as the package name and use the identifier "Object" as the name of the Ada tagged type name. Otherwise, some other consistent naming convention rules can be applied.

4.5.2 Polymorphism Perfection

Polymorphism in C++ is achieved for pointers, not for objects themselves. This is a subtle and error-prone distinction [Jorgens93]. Since polymorphism is only possible by pointers in C++, the pointer is typically converted into the class-wide access type of an Ada tagged type. This method should work in conversion but is not the best programming style in Ada. In Ada95, polymophism can be achieved by access types like C++'s pointers or class-wide types themselves. The initialization and finalization of objects in Ada is only allowed for objects, not for access types. Therefore, it seems classwide types are better than access types for polymorphism. The converted Ada programs should be modified so that polymorphism is achieved via classwide types. For example, consider a member function declared as follows:

```
void init_any_sim( simualtion* sim );
```

After the third phase of this methodology, a specification of this subprogram could be declared as follows:

```
procedure init_any_sim(sim: Simulation_ClassWide_Ptr );
```

where `Simulation_ClassWide_Ptr` is declared as follows:

```
type Simulation_ClassWide_Ptr is access all
     Simulation'class;
```

In this phase, it is recommended to change the above subprogram specification as follows:

```
procedure init_any_sim(sim: Simulation'class);
```

Of course, some change in the correspondent subprogram body may be necessary.

4.5.3 Recovering Subprogram Parameter Types

Recovering subprogram parameter types involves correcting the *subprogram profile distortion* induced during the first phase. Appendix J presents a solution for converting a C++ class member function which has a return value and also modifies its class data attributes into an Ada subprogram. During the first phase, the original C++ function must be changed into a C++ function which has no return value and the return value should be passed back by a new added additional parameter whose type is a pointer of the original value type. Then, the modified function can be converted into an Ada procedure and the new added parameter must be converted to an access type of the original value type.

However, the solution presented in Appendix J has a disadvantage. The original C++ function has a direct return value, so the return value can be directly accessed. In the

fourth phase, the C++ function is converted into an Ada procedure and the return value becomes an access type of the original value type, so the return value only can be accessed in an indirectly fashion. This is the subprogram profile distortion. After the entire application has been converted into Ada programs, this distortion should be recovered during the fifth phase. The recovering method and work are also presented in Appendix J.

4.5.4 Use of Inline Pragmas

Use of `Inline` pragmas may reduce subprogram call overhead. This work is needed only when the performance of the program is really a problem. In the first phase, the original C++ inline functions were changed into non-inline functions and their implementations were moved out of the header files. The inline functions no longer exist after the first phase. Therefore, for applications where performance is critical, these original inline functions may need to be reinserted in the fifth phase.

Ada95 provides `Inline` pragma to help programmers reduce the subprograms call overhead, but `Inline` pragmas should be used with extremely caution. If a pragma `Inline` applies to a callable entity, this indicates that the inline expansion is desired for all calls to that entity [Ada95b]. However, use of inline pragmas may violate good programming style because it may significantly increase the amount of recompilation [Cohen96] . Also, the Ada compiler may ignore the recommendation expressed by the `Inline` pragma [Ada95b]. So speeding up the execution time may not be achieved. In some cases, use of `Inline` pragmas can even slow the calling program instead of speeding it up [Cohen96]. Therefore, Cohen suggests programmers use `Inline` pragmas

once it is determined that execution time is really a problem [Cohen96]. If programmers decide to use `Inline` pragmas, they should be added in this phase.

4.5.5 Class Visibility Remapping

Class visibility remapping involves reexamining the visibility of classes in the original C++ programs and then making necessary changes to fit Ada's visibility schema. In C++, there are three levels of visibility to a class: private, protected, and public. However, the visibility of attributes and methods of an Ada tagged type only has two levels, public and private, and the visibility is determined by both the tagged type declaration and how it is placed [Jorgens93].

Section 4.3 explained why it is better to reexamine the visibility of classes and implement in Ada's visibility mechanism in this phase. Therefore, a mechanism to convert the C++ classes visibility to fit Ada's visibility mechanism is necessary.

The following rules are some guidelines to convert the C++ class visibility into the Ada visibility mechanism:

- The C++ public attributes and member functions should be implemented as public attributes and methods in the Ada package which declares the corresponding tagged type to the C++ class.
- The private member functions of a C++ class might be mapped to local Ada subprograms within the package body which implements methods of the class.
- The private data attributes of a C++ class can be mapped to local variables within the package body. The private data attributes of a C++ class can also be declared as **limited private** incomplete type variables in the public section of the package

specification and the incomplete type should be declared in the package body. An incomplete type declaration has the form "`type type_name;`". A declaration of the incomplete type will be followed by a full declaration of the named type [Cohen96].

- The protected data attributes and member functions of a C++ class can be mapped into the private data attributes and methods within the Ada package specification.

These guidelines can help in the conversion of C++ class visibility into Ada's visibility mechanism. However, simple conversion from the C++ class visibility into Ada's visibility mechanism is not recommended because the C++ class visibility results can not be achieved. To achieve the same result, the hierarchical libraries mechanism of Ada95 must be implemented in the converted Ada programs. That will be discussed in the following subsection.

4.5.6 Program Structure Reorganizing

Program structure reorganizing must take full advantage of the Ada's program structures. In this phase, all the advantages of Ada's program structures (subprogram, package, and hierarchical libraries) should be used to restructure the converted Ada programs such that the converted Ada programs are well-structured. "Well-structured programs are easily understood, enhanced, and maintained" [SPC95].

Before this phase, the converted Ada programs were still based on the C++ program structure: header file, source-code file, class, and function. The first phase of this methodology was to reorganize the original C++ software system such that the original C++ code has a style similar to Ada's package style. After the fourth phase, the converted

Ada programs only benefit from Ada's package concepts. The converted Ada programs have not yet benefited from the use of hierarchical libraries.

In Ada's hierarchical libraries mechanism, child packages allow programmers to create more precise packages with a less cluttered interface and to extend the interface as needed [SPC95]. Cohen states that child units (packages) have the following important uses [Cohen96]:

- avoiding large files and associated pragmatic problems that result from deep physical nesting
- expressing the hierarchical decomposition of a large system into subsystems
- avoiding name clashes among independently-written components of a large program
- decomposing the interface of a module into two or more parts, intended for different clients and separately specifiable in **with** clauses
- enhancing a package as a program evolves over time with minimal disturbance to existing code, including other programs that may continue to use the original definition of the package
- adding personal customizations to a shared package that will not affect other users of the package

Since Ada's hierarchical libraries have so many advantages, there is no reason not to use it. The idea of Ada's hierarchical libraries should be used to reorganize the converted Ada programs such that the converted Ada programs are well structured after this phase.

The work in this phase is critical to the quality of the converted Ada programs. The goals of the first four phases are to convert the original C++ software system into Ada

programs and not to lose any original functional requirements. The goal of this methodology is not only to convert C++ code into Ada programs but also to promote the quality, reliability, and maintainability of the converted software system. This is the real value of converting a C++ software system into Ada. Ada helps programmers to implement software engineering principles and forces programmers to use the principles of software engineering such that software systems written in Ada can be easily understood and maintained. It is a programming language for software engineers, not just for programmers. After this phase, the converted Ada programs should benefit from most of Ada's advantages. Finally, the goal of language conversion is achieved by this methodology.

5 Results and Lessons Learned

Most problems and difficulties in this thesis work occurred because mixed language programming is more different than typical single language programming. Single language programmers only need to understand one language mechanism, while mixed language programmers not only need to understand two language mechanisms, but also have to figure out the strengths and weaknesses of each language and identify the subtle differences between C++ and Ada95. Both languages are complicated. To completely understand one of them is difficult; to understand two is a great task. To use the two together is a greater task yet.

Another difficulty in mixed language programming is the lack of mixed language programming knowledge and experience. Traditionally, software programmers are trained to program in a single language development environment. They only have experience and knowledge in single language programming. The documents provided by vendors usually only provided the information for single language developers not for mixed language programmers.

In this thesis work, the problems encountered were quite different from the problems of traditional single language programming. These problems are presented in the following sections.

5.1 Converted Results

There are two main results of this thesis work. The first result is the interface package specifications created to interface the existing ObjectSim and ObjectManager

classes and RDT classes. Since these interface package specifications were successfully created, the incremental language conversion was feasible in this thesis work. These interface package specifications are listed in Appendix G.

Another important conversion result of this thesis work is the converted main program. The original C++ main program in file rdtApplication.cc was converted into an Ada procedure, named rdtMain in file rdtMain.adb. This result was produced by following the first four phases of the methodology presented in Chapter 4. From the success of converting the main program, this thesis work has shown that the methodology is feasible and practical. Even though the last phase of the methodology was not completed, this should not be an argument against the feasibility of the methodology because the last phase is an embellishing effort and it will not affect the execution results. It promotes the quality of the converted software system.

No other C++ source-code modules were converted into Ada programs. The reasons why not all RDT modules were converted into Ada95 are listed as follows:

- In some cases, Adabindgen was unable to convert some C++ header files into Ada package specifications. Regarding GNAT's C++ low-level interface Dr. Comar stated, "Our goal is not to make the process of interfacing C++ code as easy as possible to the user. ... we assume that the interfacer has some knowledge about the internals of the C++ compiler. The kind of interface we provide is more likely to be used by third-party tools specialized in interface binding generation than by direct users" [Comar]. Adabindgen is the tool that can help programmers use the GNAT's C++ low-level interface. However, Adabindgen is still under development. The alpha version of

Adabindgen, used in this thesis work, has some deficiencies. Therefore, some important C++ header files could not be converted into Ada package specifications by Adabindgen. Because of this, a lot of time was spent manually creating and debugging the interface package specifications.

- A debugging tool for mixed language programming was not available. SGI provided a debugging tool, called "cvd" to help programmers debug their programs. Originally, this tool was designed for C/C++ programmers. It provided no functions to help mixed language programmers debug their programs. Programmers can not use cvd to trace and examine their mixed language programs. It was a painful debugging experience. Cvd was upgraded recently so that it could debug C/C++ and Ada programs together. However, it was too late to be of assistance in this thesis effort.

Due to the above reasons, not all RDT modules were converted into Ada95. Appendix H lists all RDT source-code modules. This gives programmers a work-list to know how many modules need to be converted in the future.

5.2 The Differences between Single and Mixed Language Programming

In typical single language programming, programming errors usually involve syntax and semantics errors. Unlike single language programming, mixed language programming involves less time on debugging syntax and semantics errors. Basically, doing language syntax conversion is not difficult. Also, it is not necessary to understand the details of software systems and then fix semantics errors because the original software has been verified and functions well. In incremental language conversion, interface pragmas are used to interface the existing code. Using these interface pragmas is not

difficult and the existing code has been verified before. Therefore, for mixed language programming, less time is spent fixing semantics errors, but the unresolved linking symbols and run-time exception errors take most of the time.

5.3 Linking Problems

Many problems occur in the linking phase and during run-time. After the third phase of the incremental language conversion method presented in Chapter 4, mixed language development proceeded. At this time, many linking problems occurred. In single language programming, debugging information from the compiler aids in fixing syntax errors. Usually, compilers provides the information that indicates which line has the syntax error and the reasoning. Unlike compilers, linkers usually only show the names of unresolved symbols without other detailed information. Therefore, it is necessary to manually find the object file modules which define the unresolved symbols.

The main problem in the linking phase is to solve unresolved symbols. The causes of unresolved symbols are difficult to identify. Some causes are summarized in the following subsections.

5.3.1 Incorrect Manually Mangled Names

The most probable cause of unresolved symbols is that incorrect manually mangled names are provided, and are used to interface the C++ functions or variables from Ada programs. In GNAT's C++ low-level interface capabilities, the key to interfacing C++ from Ada95 is to use low-level link symbols. These link symbols are supposed to be provided by an automatic tool for interfacing C++ code from Ada95. SGI's Adabindgen is

the tool used to achieve this interface requirement. However, Adabindgen is only an unpublished product and still under development. Some deficiencies of Adabindgen are listed in Appendix F. In some cases during this thesis work, it did not correctly convert the C++ header files into Ada package specifications. This can also lead to human error in finding the correct manually mangled name. Therefore, some unresolved symbols frequently appear.

In most cases during this thesis work, the linker only gave the link names of some unresolved linker symbols. Not much information was provided to solve the unresolved symbols. The way to solve unresolved symbols is to locate the unresolved symbols by using **nm** command to thoroughly search among object files. Then check if the manually mangled name is correct. If it is not correct, then it must be replaced by the correct name. The linker usually cannot provide any information about this, so other tools must be used. For example, **grep** command can be used to search and locate where the objects were interfaced and **nm** command can be used to examine the manually mangled names. The compiler and linker provide little assistance. Experience is the most important factor in solving these kinds of problems.

5.3.2 System Libraries

If the unresolved symbols are not related to any objects of the programs, the cause of these unresolved symbols may be that some system-provided libraries are not linked properly. In traditional single language software development processes, only one language linker is used to link all separately-compiled software modules. However, there are two linkers that can be chosen in mixed language programming. The decision must be

made as to which linker should be chosen to link all mixed language modules. However, not much information is documented to assist in this decision. According to the experience of this work, the C++ linker, ld, usually is the correct linker for interfacing C++ code from Ada programs. An incorrect choice of linker may not only cause unresolved symbols in the linking phase, but also may cause exception errors during run-time. This exception error problem will be presented in Section 5.4.

Even if the correct linker has been chosen, unresolved system library symbols may still appear during the linking phase. This is because the system libraries may not be linked together with the object files. In the single language development case, the language linker usually will automatically refer to some default system libraries without explicit command options when all separately compiled software modules are linked together. However, in mixed language software development, the chosen linker may not know the other language's system libraries. Therefore, it cannot automatically link another linker's libraries. It is necessary to explicitly tell the chosen linker where another language's system-provided libraries are. Unfortunately, this kind of information is not usually documented for software developers. Since programmers may be not familiar with the unresolved symbols defined by system-provided libraries, solving these unresolved symbols is difficult for novice programmers. These kinds of problems must be fixed in a trial-and-error fashion. Thorough searching in all system libraries may be needed to identify which libraries define the unresolved symbols.

5.3.3 Incorrect Linking Sequence

Incorrect linking sequences of libraries or object files may also cause unresolved symbols. During this research effort, different linking sequences of libraries or object files were shown to have different link results. Some specific incorrect linking sequences caused unresolved symbols. The reason for this is still unclear. This problem may be related to the internal behavior of the linker, however, no documents have been found to substantiate this theory. Fortunately, the original Makefile provided the correct linking sequence of the object files and libraries.

5.4 Run-time Problems

The main concern in the run-time phase is exception error problems. Debugging the exception error presents another difficulty in this thesis work. GNAT provides good error messages for syntax errors. However, for debugging exception errors, GNAT only provides minimal help. Two exception errors, Storage_Error and Constraint_Error, frequently appeared in this thesis work. The possible causes and solutions are summarized in the following subsections.

5.4.1 Storage_Error

A Storage_Error exception can be raised by elaboration of a declaration and also by a subprogram call if many subprograms are still in progress when a new one is called [Cohen96]. A Storage_Error shows up when the program runs out of memory in the pure Ada system where all code is written in Ada. In non-pure Ada systems in which the code is written in Ada and another language, it is probably due to some bad address generating a

segmentation violation [Comar96]. However, identifying the exact location in code where the segmentation violation occurs is difficult. According to this thesis work experience, three possibilities for the cause of Storage_Error exist: incorrect elaboration, uninitiated pointer, and the wrong choice of linker.

Incorrect elaboration has the highest possibility for causing Storage_Error and Constraint_Error. From the experience of this thesis work, the most likely reason for a Storage_Error is that at least one declaration is not correctly elaborated. C++ does not have elaboration, but Ada does. "In Ada, declarations are processed as they are encountered during program execution. A declaration may refer to values computed earlier in the execution. The act of processing a declaration is called *elaboration*" [Cohen96]. When Storage_Error is raised, it is difficult to locate which declaration was not properly elaborated and where the declaration is.

Most of Storage_Error problems were caused by the declarations in the created interface package specifications because the declarations were not elaborated correctly. This often caused the Storage_Error to be raised before the first execution statement executed. Before execution of an Ada main subprogram, all compilation units directly or indirectly used by the main program are elaborated [Cohen96]. Therefore, each **with** clause within the main program starts a *dependency trace*, which is a sequence of the hierarchically referred packages. Packages in the each dependency trace will be elaborated in order. Storage_Error will be raised whenever a package in any one dependency trace cannot be correctly elaborated. Therefore, it is difficult to identify which package caused Storage_Error.

The strategy to solve Storage_Error exceptions is to first narrow down the problem range and then check the declarations within the package to locate the exact cause of elaboration error. One way to narrow down the problem range is to identify which one interface package specification causes the problem. Then each declaration in the identified package can be examined to find out which declarations were wrong.

To identify which package causes Storage_Error, the system dependency diagram is helpful. A *system dependency diagram* is the diagram which describes the dependencies among software modules in the system. In a system dependency diagram, each node represents a software module and each directed edge indicates the referring relationship. The system dependency diagram provides all dependency traces and the packages in the trace. This can assist in understanding the sequences of package elaboration, so all directly or indirectly referred packages can be traced down.

However, this method only helps the problem range to be narrowed down to a single package. The real cause of Storage_Error has not yet been identified. "Elaboration of a package declaration entails elaborating all the declarations within it, in order" [Cohen96]. It is necessary to trace down which declaration(s) caused the Storage_Error.

The second possible cause of Storage_Error is that a pointer is not correctly initialized. Pointers should be initialized to an address, **0,** or **`NULL`** either when they are declared or in an assignment statement [Deitel94]. In pure C++ software systems, sometimes, an uninitiated pointer may not cause any run-time error. However, this does not mean that there will not be any run-time errors in mixed language programming. An uninitiated pointer is a dangerous time bomb in programs because it may point to an

address out of the users program segment and then cause the Ada run-time system to raise Storage_Error.

The third possible cause of Storage_Error is the wrong choice of linker. The wrong choice of linker may not only cause unresolved symbols in the linking phase (see Section 5.2.2), but may also cause Storage_Error in run-time. This is a very difficult problem to debug. After programs have been compiled and linked without showing any error messages, programmers usually will not think anything is wrong in the linking phase because Storage_Error was raised during run-time. However, as was discovered during this thesis work, it is possible to successfully link all programs by using the wrong linker, but problems may occur later in the run-time phase. The natural response in such a sitiulation is to examine the source code and try to figure out why Storage_Error was raised. Because the linker did not show any error messages, it is unlikely that the linker would be suspected. Appendix E presents such a case which occurred in this thesis work. In this case, programs were compiled and linked without any error messages. However, Storage_Error was still raised during run-time. Finally, after receiving expert assistance from Dr. Cyrille Comar of AdaCore, Inc., the programs were successfully built by the correct linker using the proper building procedures shown in Appendix E.

5.4.2 Constraint_Error

Constraint_Error is another common run-time error that occured during this thesis work. Cohen summarized many causes of Constraint_Error in [Cohen96]. However, in this thesis work, the two most probable causes of Constraint_Error were (1) elaboration errors and (2) the mismatch between the converted Ada tagged record with the original C++

class declaration. These two causes usually appeared in the created Ada interface package specifications. If Constraint_Error is raised, the interface package specifications should be the first place examined.

Incorrect elaboration may also cause Constraint_Error. Section 5.3.1 stated that incorrect elaboration may cause Storage_Error. Additionally, it may also cause Constraint_Error. Therefore, the same strategy presented in Section 5.3.1 is also useful in solving Constraint_Error.

During this thesis effort, most Constraint_Errors occurred because the original C++ class failed to be correctly converted into the Ada tagged type. Two common errors in this area were (1) the types of the C++ class data attributes were not correctly converted into the parallel data types in the corresponding Ada tagged record type or (2) a C++ class without any virtual function was converted to an Ada tagged type. Because both errors may cause the memory layout of the C++ class to be different from the Ada tagged type's memory layout, Constraint_Error may be raised in run-time.

A C++ class is converted a tagged type and data attributes of the class are converted the components of the tagged type. If the type of any one data attribute within the C++ class was not correctly converted into its parallel data type within the Ada tagged type, the C++ class is incompatible with the converted Ada tagged type. Section 3.1.4 states that only safe types can cross language boundaries. In GNAT, only the types listed in Table 8 can be viewed as safe types if they are correctly converted into the parallel types, because GNAT will guarantee the correspondence between Ada types and the parallel data types in C/C++ [SGI]. Therefore, if the type of any one data attribute within the C++ class was not

correctly converted its parallel data type within the Ada tagged record type, the C++ class becomes an unsafe type to cross language boundaries. Therefore, when the incompatible parallel-typed component of the Ada tagged record is assigned, the Ada run-time system may detect that the assigned value cannot fit the type of the mismatched component and will raise Constraint_Error.

In GNAT's C++ low-level interface capabilities, only a C++ class having at least one virtual function should be equally converted to an Ada tagged type. Otherwise, the memory layout of the Ada tagged type will be different than the memory layout of the original C++ class. This is because a C++ class having no virtual functions does not have a virtual table to associate with but an Ada tagged type does. *Virtual table* is an internal table created by the compiler that contains the addresses of the virtual member functions. The dynamic dispatching capability, one of the object-oriented programming (OOP) features, is achieved by using this table. The detailed function of virtual tables is beyond the scope of this thesis.

6 Conclusion and Future Study

6.1 Accomplishments

This thesis presented an incremental language conversion method for converting C++ programs into Ada95 programs. The methodology consisted of five phases: (1) reorganizing the software application, (2) breaking the mutual dependencies, (3) creating necessary package specifications to interface the existing C++ classes, (4) converting C++ code into Ada programs, and (5) embellishment. This methodology has the following advantages:

1. The converted programs have more Ada language advantages than "C++ style" Ada programs. Simply translatting C++ programs into Ada may not add any value to the programs. This methodology not only correctly converts C++ applications into Ada95 programs, but also takes advantage of many of Ada's language features which support good software engineering principles. This methodology outlines a way to convert the original C++ application into a well-structured Ada program such that the converted Ada program has several advantages which did not exist in the original C++ application.

2. Detailed application knowledge is not necessary. Only a high level knowledge of the software system is needed. Software engineers do not need to do reengineering work nor understand the details of the software system. According to the experiences of this thesis work, language conversion can be accomplished without understanding the details of the software system.

3. Incremental conversion appears to be a more straight forward process than non-incremental conversion. Following the phases of this methodology, software modules can be converted one at a time. Therefore, it is easier to debug and verify the functions of each module than for an entire system. Also, the results after each phase are verifiable. In this methodology, immediate results can always be obtained. So, the risks of language conversion can be reduced as much as possible.

Another important accomplishment of this thesis work was that the RDT main program was successfully converted into Ada95. The real value of this accomplishment is not just the Ada source code of the main program, but that the converted RDT main program has shown that this methodology is feasible and practical. Also, based on the experience and knowledge learned from converting the RDT main program, it should be easier to convert other RDT modules.

6.2 Future Study

Although an incremental language conversion methodology has been successfully developed in this thesis work, some remaining work needs to be done. The following discussion of future work is broken into three subsections. The first subsection recommends finishing the conversion of RDT. The second subsection recommends verifying all created ObjectManager and ObjectSim Ada package specifications. The third subsection recommends to study the other unaddressed C++ features.

6.2.1 Convert All RDT Modules

Since only the RDT main program has been converted into Ada95, the first recommendation for future study is to convert the other RDT C++ modules. Appendix H lists all RDT source-code modules. This gives programmers a work-list to know how many modules need to be converted in the future. After the entire RDT is converted into Ada95, the original RDT and the converted RDT should be compared in the areas of performance, maintainability, reliability, and size.

Also, more conversion experience from C++ to Ada should be pursued. Knowledge obtained from this effort could produce a guidebook to make future conversion work easier. As programmers gain more experience in the conversion process they will learn and understand the subtle differences between C++ and Ada95.

6.2.2 Verify the Created Ada Package Specifications

In order to interface ObjectSim and ObjectManager, some C++ header files used by RDT were converted into Ada package specifications. However, other C++ header files, which were not used by RDT, have not been converted into Ada package specifications yet. For future projects, other header files should be converted into Ada package specifications. Therefore, a complete interface for Ada programmers to interface ObjectSim and ObjectManager should be undertaken.

Also, some created Ada package specifications have not been verified yet, because they are not used by RDT. Some potential problems may occur when they are used to interface ObjectSim and ObjectManager. In ObjectSim and ObjectManger header files,

only the header files used by RDT have been verified. These package specifications are listed in Appendix G.

6.2.3 Unaddressed C++ Features

Some C++ features were not addressed in this thesis work because they were either not used in RDT or they are used in RDT modules that were not converted during this effort. Therefore, the methods to convert these C++ features into Ada were not included in this thesis. Unaddressed C++ features are *templates*, *friend classes*, *exception handling*, *file processing*, and *stream I/O*. These C++ features should be studied in the future so the methods to achieve the same effects in Ada95 can be found.

Bibliography

[Ada95a] The Ada9X Design Team Rationale, Ada 95 Rationale, Intermetrics, Inc., Cambridge, MA, January 1995.

[Ada95b] The Ada9X Design Team Rationale, Ada 95 Reference Manual, Intermetrics, Inc., Cambridge, MA, January 1995.

[APR93] Ada 9X Project Report, Introducing Ada 9X, Office of the Under Secretary of Defense for Acquisition, Washington, DC 20301, May 1993.

[Balfour2] Brad Balfour, "Ada95 Tips and Tidbits Number 2: Expressing Design Inheritance Relationships in Ada 95"

[Balfour3] Brad Balfour, "Ada95 Tips and Tidbits Number 3: Inheritance & Child Library Units"

[Cernose93] G. J. Cernosek, "ROMAN-9X: A Technique for Representing Object Models in Ada 9X Notation", TRI-Ada '93 Conference Proceedings, p385-406, Seattle, Washington, Sep. 18-23 1993.

[Cohen96] Norman H. Cohen, Ada as a second language, The McGraw-Hill Companies, Inc., 1996.

[Comar] Cyrille Comar, GNAT/C++ Low-Level Interface, in ftp://wuarchive.wustl.edu/languages/ada/compiler/gnat/distrib/docs filename: c++-interface.ps.gz.

[Deitel94] H. M. Deitel and P.J. Deitel, C++ How to Program, Prentice Hall, Englewood Cliffs, NY, 1994.

[Dewar94] Robert Dewar, "The GNAT Compilation Model", Proceedings of the TRI-Ada '94 Conference, Baltimore, Maryland, 1994, ACM.

[Gannon77] J. D. Gannon, "An Experimental Evaluation of Data Type Conventions", CACM, v20, no. 8, Aug. 1977.

[Gardner93] Michael Thurma Gardner, A Distributed Interactive Simulation Based Remote Debriefing Tool for Red Flag Missions, MS thesis, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH AFIT/GCS/ENG/93-09, December 1993.

[Garlan93] David Garlan and Mary Shaw, "An Introduction to Software Architecture", Advances in Software Engineering and Knowledge Engineering, Vol. I, World Scientific Publishing, 1993.

[Garlan95] David Garlan and Dewayne Perry, "Introduction to the Special Issue on Software Architecture", IEEE Transaction on Software Engineering, Vol. 21, No. 4, p269-274, Apr. 1995.

[Gart95] M. Gart, "Interfacing Ada to C - solutions to four problems", Proceedings of the TRI-Ada '95 Conference, Anaheim, CA., November 5-10, 1995, ACM.

[HBAP] Home of Brave Ada Programmers, "An Introduction to Ada", http://lglwww.epfl.ch/Ada/Introduction.html Nov. 1995..

[Horton85] Michael J. Horton and Teri F. Payton, "Integrating Ada into Multi-lingual Systems: Issues and Approaches", Proceedings of the 3rd Annual National Conference on Ada Technology, 1985.

[Hughes91] David K. Hughes, "Multilingual Software Engineering Using Ada and C", Software Engineering Notes, vol. 16, no. 4, ACM SIGSOFT, Oct. 1991.

[Johnston] Simon Johnston, Ada-95: A guide for C and C++ programmers, http://lglwww.epfl.ch/Ada/Ammo/Cplpl2Ada.html.

[Jorgens93] Jesper Jorgensen, "A Comparison of the Object Oriented Features of Ada 9X and C++", Lecture Notes in Computer Science 688, Ada-Europe '93, Springer-Verlag, Berlin, 1993.

[Kapur95] S. Kapur, "Ada and C Interface Issues in the Development of Peripheral Device Support Libraries", Proceedings of the TRI-Ada '95 Conference, Anaheim, CA., November 5-10, 1995, ACM.

[KSC93] Kaman Science Corporation, A Review of Non-Ada to Ada Conversion, in World Wide Web Site : http://www.utica.kaman.com/techs/ada/ NonAda2Ada.ToC.html, 1993.

[Lippman91] Stanley B. Lippman, C++ Primer, Addison-Wesley, Reading, MA, 1991.

[Martin95] Robert C. Martin, Designing Object-Oriented C++ Application Using The Booch Method, Prentice-Hall, Englewood Cliffs, NY, 1995.

[Pohl93] Ira Pohl, Object-Oriented Programming Using C++, The Benjamin/Cummings Publishing Company, Inc., 1993.

[Quiggle96a] Thomas J. Quiggle, "Re: Thesis work", Electronic Mail Message, Silicon Graphics, Inc. Mountain View, CA, 28 Feb 1996.

[Rosen95] J. -P. Rosen, "A naming convention for classes in Ada 9X", ACM Ada Letters, XV(2):54-58, Mar.-Apr. 1995.

[Rumbau91] James Rumbaugh and others, Object Oriented Modeling and Design, Prentice Hall, 1991.

[Schonb94] Edmond Schonberg, Bernard Banner, "The GNAT Project: A GNU-Ada 9X Compiler", Proceedings of the TRI-Ada '94 Conference, Baltimore, Maryland, 1994, ACM.

[SGI92] Silicon Graphics, Inc. Graphics Library Programming Guide, Volume I. Silicon Graphics, Inc., Mountain View, CA, 1992.

[SGI94] Silicon Graphics, Inc. IRIS Performer Programming Guide (version 1.2) Silicon Graphics, Inc., Mountain View, CA, 1994.

[SGI96] Silicon Graphics, Inc. GNAT Release Note (Electronic copy in Release Note) Silicon Graphics, Inc., Mountain View, CA.

[SGI] Silicon Graphics, Inc., GNAT Reference Manual, (Electronic copy in *Insight*) Silicon Graphics, Inc., Mountain View, CA.

[Snyder93] Mark I. Snyder, ObjectSim - A Reusable Object Oriented DIS Visual Simulation, MS thesis, AFIT/GCS/ENG/93D-20. School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB OH, December 1993.

[SPC95] Software Productivity Consortium, Ada95 Quality and Style: Guidelines for Professional Programmers, SPC-94093-CMC, Herndon, Virginia, Oct. 1995.

[Taft94] S. Tucker Taft, "Multiple Inheritance in Ada 9X", Ada 9X Language Study Note, No. 1033, 1994.

[Uhde95a] Kimberly K. Uhde, Translating C++ Software into Ada95, Technical Report 95-03, Computer Science Department, Vanderbilt University, Nashville, Tenn., 1995.

[Uhde95b] Kimberly K. Uhde, Translating C++ Software into Ada95, CrossTalk, published by Software Technology Support Center, vol. 8, No. 10, Oct 1995.

Appendix A. The RDT Modules and Their Sizes

This appendix lists the major modules of RDT. Some C code modules which are used to display forms and deal with user interfaces are not included in this section because they were not converted into Ada in this thesis work. Also some AFIT developed libraries, which are for other projects not only for RDT, are not listed in this section.

Table A- 1. RDT modules and their sizes

File Name	Size[2] (LOC)	Classes	Functions
rdtColorFunctions	600+23[3]		buildColorTable setRDTColors getGraphicsConfig saveColors restorColors resetColorTable buildMsnColorTable getMsnColorIndex
rdtConfig	140+27		setConfiguration quitVersionCb showInfoCb configureViewsCb showErrorMessage
rdtErrorMessage	98+67		quitErrorMessage show ErrorMessage

[2] The size of a software module inclues the comments in the source code.

[3] The first number indicates how many lines in the source code file and the second number indicates how many lines in the header file.

Table A-1. RDT modules and their sizes (Cont'd)

File Name	Size (LOC)	Classes	Functions
rdtController	2280+68		accept3DAC acceptAircraft acceptAircraftColumn acceptPairData activateKeypad deactivateKeyPad DeadReckoningCb eraseExerciseCb expandPositionerCb hideCallSignsCb hideFlightPathsCb hideGeoLabelsCb hideLowActivityCb hideWeaponCb keypadEnterCb loadACStateFormCb loadCentroidFormCb pageDataViewCb printShStateCb quitControlFormCb quitExpandPlanViewCb quitStartFormCb refreshExerCb refreshSummaryCb resetAircraftColumnCb resetPairDataViewCb select3DACCb select3DViewCb selectAircraftCb selectDataViewCb selectPairViewCb setAircraftColumnCb setCentHdgCb setEngineerDataViewCb setEventLoggingCb

Table A-1. RDT modules and their sizes (Cont'd)

File Name	Size (LOC)	Classes	Functions
rdtApplication	2203+53	RDT_Terrian	public: configure_channel
		RDT_View	draw
		RDT_Forms_View	public: init_draw pre_draw draw
		RDTViewPlayer	public: getScaleFactor drawWpnEvents
		CentroidPlayer	public: init propogate draw setView findMidpoint getXYDistance
		PlanViewPlayer	public: init propogate draw init_draw
		TetherPlayer	public: init propogate draw moveDetach OldTime
		CockpitPlayer	public: init propogate draw
		RDT_App	init_sim init_draw_thread pre_draw post_draw propogate showDataViews showPairData

Table A-1. RDT modules and their sizes (Cont'd)

File Name	Size (LOC)	Classes	Functions
rdtCentroid	213+15		selectCentroidAircraftCb acceptCentroidAircraft setCentroidPlayerCb exitCentroidFormCb resetCentroidACCb
rdtEventManager	375+42	RDTEventManager	public: update_state initialize alloc_shared
rdtCirQueue	188+50	CircleQueue	public: init reset addQueue startIterate iterateBackward printQueue
rdtEventQueue	238+71	EventQueue	public: initialize addQueue startIterate iterateBackward printQueue
rdtGeoRefMgr	1115+92	GeoRefManger	public: initilaize drawGeoRefs convertLLtoXYZ showData non-public: Geodetic_to_Geocentric Geocentric_to_ Geodetic
rdtHelp	178+24		load_it startHelpCb quitHelpCb showHelpCb
rdtUtilities	694+30		computePairData validatePairDataAC validateDataViewAC calcPairInfo drawPlayerInfo buildTypeTable

Table A-1. RDT modules and their sizes (Cont'd)

File Name	Size (LOC)	Classes	Functions
rdtUserInterface	1309+44	UserInterface	public: initialize assignShared delayUntil readForms showData showPairs shutDown
rdtNetManager	510+75	RDTNetManager	public: get_net update alloc_players init protected: alloc_shared
rdtREU	729+233	rdt_Round_ Earth_Utils	public: round_flat_xforms euler_angles_from_matrix eeng_euler_angles_from_matrix matrix_from_euler_angles singularity_matrix_from_euler_angles euler_to_pfmr_hpr pfmr_to_euler_hpr rnd_euler_to_flat singularity_rnd_euler_to_flat flat_euler_to_rnd singularity_flat_euler_to_rnd rnd_pos_to_flat flat_pos_to_rnd rnd_vec_to_flat flat_vec_to_rnd alt_msl ECI_to_ECCF ECCF_to_ECI reference_frame_mod_ftr reference_frame_mod_rtf core_segment lat_long_from_xyz private: rount_flat_xforms_2 flat_pos_to_rnd_2

Table A-1. RDT modules and their sizes (Cont'd)

File Name	Size (LOC)	Classes	Functions
rdtSetRLE	498+22		setRLE drawRLE pushRLE popRLE reDrawRLE
rdtStrFnct	158+20		strinsert strpartfill my_itoa
total			
18	11767+995 =12762	17	80

Appendix B. Reorganized RDT and ObjectSim Software Modules

B.1 Reorganized ObjectSim Software Modules

In this phase (Reorganizing Software Application), the main work for ObjectSim was to strictly separate implementation from class declaration. Some inline member functions were moved into their source-code files. The C++ inline member functions are the member functions whose implementations are in header files. The following work was done in reorganizing the ObjectSim software system:

1. The inline functions of the Modifier class, `init_state` and `reset`, were moved into the new created source-code file, called modifier.cc.

2. The inline function of the `Terrain` class, `clamp`, was moved into the new created source-code file, called terrain.cc.

3. The inline function of the `Attachable_Player` class, `draw`, was moved into the new created sourced-code file, called attachable_player.cc.

4. The constructor of the `Round_Earth_Utils` was modified because of one compiler problem. The old source code was not fully compatible with the current C++ language. Early C++ language allowed memory management for objects to be controlled by casting a specified type to modify **this** pointer [Pohl93]. However, new C++ does not allow programmers to modify **this** pointer. In this conversion case, there was one class whose constructor was allocated using dynamic memory allocation. In this way, **this** pointer was modified to point to a memory location allocated by memory management.

This is not allowed in new C++ languages. The solution was simply taking away the dynamic memory allocation and just using default constructor.

5. The inline functions of the `Simulation` class, `alloc_shared`, `init_draw_thread`, `pre_draw` and `post_draw`, were moved into the source-code file simulation.cc.

B.2 Reorganized RDT Software Modules

The original RDT main program module, rdtApplication, was dissected into several modules such that each module encapsulated only one C++ class. The original RDT software module `rdtApplication` was dissected into nine new created modules: `CentroidPlayer`, `CockpitPlayer`, `PlanViewPlayer`, `TetherPlayer`, `rdtApp`, `rdtTerrain`, `rdtView`, `rdtViewPlayer`, and `rdtFormsView`. The file names were named according to the class names they encapsulated. In the first phase (Reorganizing Software Application), the original module-wide global variables declared in file rdtApplication.cc were moved to file rdtApp.h. The other system-wide global variable declarations were listed in rdtGlobals.h as before. The source-code file of each class has not been changed at all. Each class still has its original function. The following table, Table , shows the new created software modules and their sizes.

Table B-1. New created RDT software modules and their sizes

Module Name	Size (LOC)
CentroidPlayer	33+453=486
CockpitPlayer	23+195=218
PlanViewPlayer	17+234=251
TetherPlayer	17+321=338
rdtApp	59+350=409
rdtTerrain	11+17=28
rdtView	14+18=32
rdtViewPlayer	22+204=226
rdtFormsView	19+94=113
Total	2101

Appendix C. Breaking the Mutual Dependencies of ObjectSim

C.1 Simplified System Dependencies Diagram

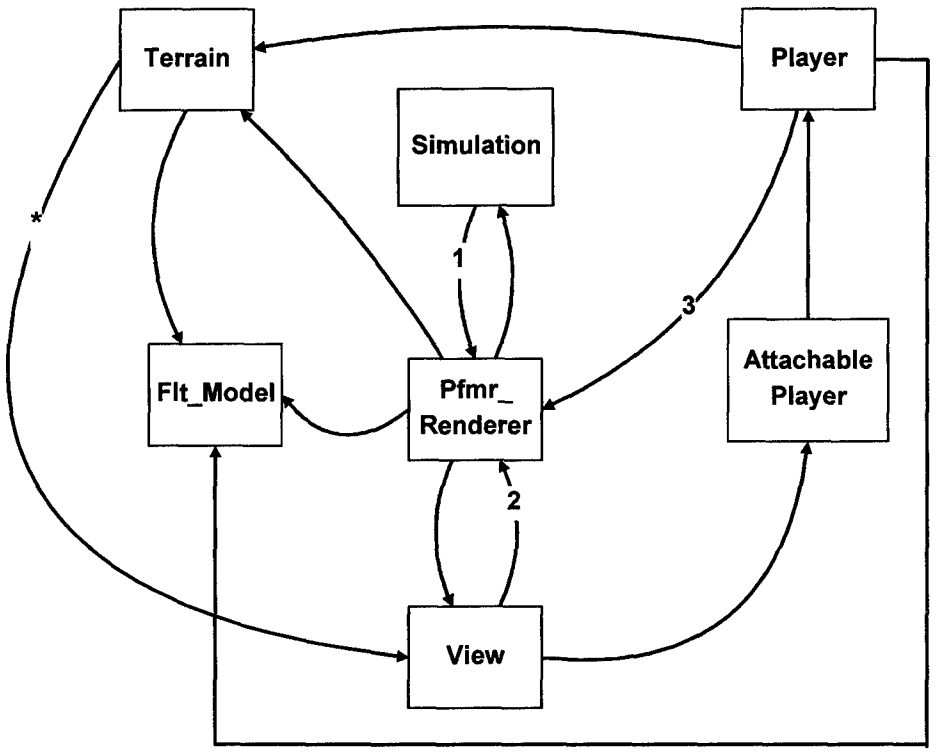


Figure C-1. The simplified ObjectSim system dependencies diagram

The above diagram, Figure C-1, is the simplified system dependencies diagram[4] of ObjectSim. There are two directed links remarked by "*". Originally, the header file of `Terrain` class did include the header file of `View` class. However, this inclusion in the original ObjectSim source code was not needed. Simply removing these two inclusions without any other change can solve the mutual dependency between `Terrain` and `View`. Therefore, these two inclusions have been taken away.

There are three loops in this ObjectSim simplified system dependencies diagram: (1) `Simulation` → `Pfmr_Renderer` → `Simulation`, (2) `Pfmr_Renderer` → `View` → `Pfmr_Renderer`, and (3) `Player` → `Pfmr_Renderer` → `View` → `Attachable_Player` → `Player`. These three loops were the mutual dependencies in ObjectSim. The following sections will describe how to break these three mutual dependencies and the rationale.

C.2 Breaking the First Mutual Dependency

Table C- 1. The mutual dependency between Simulation and Pfmr_Renderer

Simulation	Pfmr_Renderer
`#include "pfmr_renderer.h"` `:` `:` `class Pfmr_Renderer;` `class Simulation {` `public:` `<< Abstract member function declaractions >>` `Pfmr_Renderer* Renderobj;` `:` `};`	`#include "simulation.h"` `:` `:` `class Pfmr_Renderer {` `public:` `void init(Simulation* theapp, ...);` `<< Other member function declaration >>` `};`

Table C-1 shows how `Simulation` and `Pfmr_Renderer` depend upon each other. `Simulation` needs to include `Pfmr_Renderer` because it wants to know the `Pfmr_Renderer` object associated with it. Also `Pfmr_Renderer` needs to include

---

[4] This simplified system dependencies does not contain the system libraries header files and some constants definition header files.

Simulation because one of its member functions, called init, needs to pass in a Simulation object. There are two ways to break this mutual dependency: (1) taking off the association with Pfmr_Renderer from Simulation or (2) removing the Simulation parameter of init, which is a member function of Pfmr_Renderer. It seems the first one is the better option in this case. After checking the source-code file of the Pfmr_Renderer, shown in Figure C-2, software engineers can find out if the ripple effects of the second option will scatter over the four member functions of Pfmr_Renderer.

```
Typedef struct {
  ........
  Simulation*    linkapp;
  .....
} shared;
static Shared*  Sh;
void Pfmr_Renderer::init(Simulation* theapp,
                         int numpipes,
                         Terrain* theterrain)
{
       :
       :
       Sh->linkedapp = theapp;
       :
       :
       Sh->linkedapp->alloc_shared();
       :
       :
       Sh->linkedapp->init_sim();
       :
       :
}
void Pfmr_Renderer::render()
{
     :
     :
     Sh->linkedapp->propogate(exitflag);
     :
}
  void DrawChannel(........)
{
     :
     :
     Sh->linkedapp->pre_draw();
     :
     Sh->linkedapp->post_draw();
}
  void Openpipeline(....)
{
     :
     Sh->linkedapp->init_draw_thread();
}
```

Figure C- 2. The simplified source-code of Pfmr_Renderer class

On the other hand, the first option only needs to remove the attribute from `Simulation` to `Pfmr_Renderer`. Since there is only one `Pfmr_Renderer` object in the applications of ObjectSim, programmers can declare a application-wide global variable for `Pfmr_Renderer` objects. Doing it this way, `Simulation` class still can refer to the `Pfmr_Renderer` instead of implementing the association as an attribute. Actually, most applications of ObjectSim did declare an application-wide global variable to refer to it. Therefore, the first option will have less ripple effects than the second one. The only modification is that the inclusion of pfmr_renderer.h in `Simulation` and the data attribute, `Rendererobj`, have been taken off.

Because of this modification, the applications of ObjectSim need be changed to fit themselves into the modified ObjectSim. All ObjectSim applications need to declare an application-wide global variable to refer to the `Pfmr_Renderer` object. All references to `Pfmr_Renderer` should be done by accessing this global variable instead of indirectly refering to `Pfmr_Renderer` through `Simulation`.

C.3 Breaking the Second Mutual Dependency

The second mutual dependency is between Pfmr_Renderer and View. Table C-2 shows how this mutual dependency happened. `Pfmr_Renderer` needs to include `View` because it wants to know which Views are associated with it. On the other hand, `View` needs to include `Pfmr_Renderer` because it wants to knows the Pfmr_Renderer object that is associated with it. Assuming that the inclusion of `View` in `Pfmr_Render` is taken away, the impact is that two member functions, `arbitrate` and `toggle_view`, need to be modified and a new mechanism is needed to record all assigned Views.

Table C- 2. The mutual dependency between `View` and `Pfmr_Renderer`

View	Pfmr_Renderer
`#include "pfmr_renderer.h"` `:` `:` `class View {` `public:` `:` `<< Abstract member function declarations >>` `:` `Pfmr_Renderer* Renderobj;` `:` `:` `};`	`#include "view.h"` `:` `:` `class Pfmr_Renderer {` `public:` `void arbitrate(View* theview, ...);` `void toggle_view(View* theview, ...);` `:` `:` `<< Other member function declarations >>` `View** assigned_view;` `};`

As in the first mutual dependency, it seems that to remove the `View`'s association `Pfmr_Renderer` attribute is a better choice. Therefore, the inclusion of `Pfmr_Renderer` in `View` is removed. The ripple effect of this modification is listed as follows:

(1) The applications of ObjectSim need to declare an application-wide global variable to refer to the `Pfmr_Renderer` object. Actually, this is not extra work because it is also necessary to break the first mutual dependency.

(2) Since the `Renderobj` attribute of the `View` class has been removed, the statements in the source code of `View` that access `Renderobj` should be removed. That means lines 12, 14, 24, and 25 in Figure C-3 must be removed.

```
1   void View::new_view(int desired_pipe)
2   {
3   (*MyPipe) = desired_pipe;
4   float far = 475000.0f;
5   scene = pfNewScene();
6   nullroot = pfNewGroup();
7   My_Model = new Flt_Model();
8   My_Model->RotDCS = pfNewDCS();
9   My_Model->root = nullroot;
10  pfAddChild(My_Model->RotDCS, My_Model->root);
11  pfAddChild(scene, My_Model->RotDCS);
12  Renderobj->arbitrate(this,desired_pipe);
13  pfChanScene(chan,scene);
14  Renderobj->terrain->configure_channel(chan);
15  pfChanNearFar(chan, 0.1f, far);
16  pfChanFOV(chan, 45.0f, -1.0f);
17  terraintrans = pfNewDCS();
18  playertrans = pfNewDCS();
19  pfAddChild(scene,terraintrans);
20  pfAddChild(scene,playertrans);
21  pfChanTravMask(chan, PFTRAV_DRAW, (*curmask));
22  chanmask = (*curmask);
23  (*curmask) = (*curmask)<<1;
24  pfAddChild(playertrans,Renderobj->players);
25  pfAddChild(terraintrans,Renderobj->terrain>root);
26  }
```

Figure C-3. Program 6: The member function of the View class that involves the second mutual dependency

The second ripple effect is a bigger problem in breaking the second mutual dependency. Because of removing lines 12, 14, 24, and 25 in Figure C-3, ObjectSim's original execution has been changed. The functions at these lines must be recovered somewhere in the system. The following paragraphs present a method to minimize the ripple effect for breaking this mutual dependency.

The way to break the mutual dependency is to remove any one of the links in the mutual dependency cycle. In order to decide which directed link should be removed, all dependencies involved in the mutual dependency should be carefully examined. Each dependency implies a header file inclusion. In order to remove a header file, the reason to include the header file must be eliminated. Therefore, all statements which result in the

inclusion should be carefully examined. The statements that result in the inclusion are *dependency points* (DPs). When a module involves a mutual dependency, there exists at least one DP. If some engineering work can be done to remove all DPs in the module, then the inclusion, which involves the mutual dependency, can be removed. Then the mutual dependency can be broken. In the second mutual dependency, the statements referring to the `Renderobj` are DPs because the inclusion of `Pfmr_Renderer` in `View` can be simply removed if these statements can be removed from `View`. Therefore, lines 12, 14, 24, and 25 in Figure C-3 are DPs.

After identifying the DPs, resequencing the statements may be needed. A program works both because correct statements have been chosen to achieve the expected results and the execution sequences of the statements are correct. The `new_view` member function of `View` consists of six parts: (1) line 1 ~ line 11, (2) line 12, (3) line 13, (4) line 14, (5) line 15 ~ line 23, and (6) line 24 ~ 25. The DPs partition `new_view` into several pieces of code. In the other word, the entire result of `new_view` can be achieved by calling six functions and each function is to execute the exact code of each part. Before, ObjectSim applications only needed to call new_view once. However, if only removing DPs (line 12, 14, 24, and 25) without any resequencing, then these six different functions corresponding to each part must be called in sequence. That means the original coherent member function would became six trivial calls in sequence.

In order to keep as much of the original coherence of `new_view` as possible, resequencing and regrouping all statements of `new_view` is needed to reduce the number of partitions. To achieve this goal, all DPs must be moved as close to each other as possible. A *statement block* (SB) is an ordered set of statements such that the first

statement in a SB is right after a DP and the last statement in a SB is right before the next DP. For example, in Figure C-3, the statement 13 is the first SB and the statements from 15 to 23 are the second SB. Note: the block that contains statement 3 to 11 is not a SB because the first line of the block is not a DP.

In order to correctly regroup the statements of new_view into less partitioned parts, Figure C-4 presents a method to follow. The overall goal of this method is to move the DPs to be as close to each other as possible. These DP blocks can be replaced by a new block which function as the original functions of the DP blocks. Then the application of ObjectSim can call this new block to recover the lost functions performed by the DP block. The resequenced results after following this method is shown in Figure C-5. This means the original `new_view` can be also written as Figure C-5 without any function loss. Therefore, `new_view` can be decomposed into four parts. The first part, line 1 ~ 20, remains in `new_view`. The second and fourth part are two blocks of DPs, so these two parts must be removed to somewhere else in the applicatiion of ObjectSim to break the mutual dependency. The third part is moved to a newly created member function of `View`, named `setup_drawing`.

Moving rule:

When a statement is moved to another block, it is appended to the top or bottom of that block, depending on its relative location to the statement. For example, if the statement is moved to a block above it, it is appended to the bottom of that block. If it is moved to a block below it, it is appended to the top of that block.

Procedure:

(1) Identify the DPs.

(2) Starting from the first SB,

(2.1) Starting from the last statement in the SB, if this statement can be moved to the next block then move it. Otherwise leave it in the same position.

(2.2) Using the bottom-up approach, choose the next statement and repeat (2.1) until all statements in the SB have been checked.

(3) Go to the next SB and repeat (2) until all SBs have been processed.

(4) Starting from the last SB,

(4.1) Starting from the first statement in the SB, if this statement can be moved to the previous block then move it. Otherwise leave it in the same position.

(4.2) Using the top-down approach, choose the next statement and repeat (4.1) until all statements in the SB have been checked.

(5) Go to the previous SB and repeat (4) until all SBs have been processed.

Figure C-4. A method to resequence the original a function and keep its coherence as possible as it can

```
1   void View::new_view(int desired_pipe)
2   {
3   (*MyPipe) = desired_pipe;
4   float far = 475000.0f;
5   scene = pfNewScene();
6   nullroot = pfNewGroup();
7   My_Model = new Flt_Model();
8   My_Model->RotDCS = pfNewDCS();
9   My_Model->root = nullroot;
10  pfAddChild(My_Model->RotDCS, My_Model->root);
11  pfAddChild(scene, My_Model->RotDCS);
17  terraintrans = pfNewDCS();
18  playertrans = pfNewDCS();
19  pfAddChild(scene,terraintrans);
20  pfAddChild(scene,playertrans);
-----------------------------------------------------
12  Renderobj->arbitrate(this,desired_pipe);
14  Renderobj->terrain->configure_channel(chan);
-----------------------------------------------------
13  pfChanScene(chan,scene);
15  pfChanNearFar(chan, 0.1f, far);
16  pfChanFOV(chan, 45.0f, -1.0f);
21  pfChanTravMask(chan, PFTRAV_DRAW, (*curmask));
22  chanmask = (*curmask);
23  (*curmask) = (*curmask)<<1
-----------------------------------------------------
24  pfAddChild(playertrans,Renderobj->players);
25  pfAddChild(terraintrans,Renderobj->terrain->root);
26  }
```

Figure C-5. The resequenced result of `new_view`

The ripple effect of this modification is that the applications of ObjectSim need to add the following piece of code whenever they call `new_view`:

```
Renderobj->arbitrate(this,desired_pipe);
Renderobj->terrain->configure_channel(chan);
setup_drawing();
pfAddChild(playertrans, Renderobj->players);
pfAddChild(terraintrans,Renderobj->terrain->root);
```

where `Renderobj` is the application-wide global variable used to refer the Pfmr_Renderer. For example, the RDT main program in file rdtApplication.cc should be modified like the program shown in Figure C-6.

```
main (int argc, char *argv[])
{
      :
      :
  LeftView.new_view(0);
  // New code for recovering the lost function because of
  // breaking the mutual dependency.
  Renderer.arbitrate(&LeftView,0);
  Renderer.terrain->configure_channel(LeftView.chan);
  pfAddChild(LeftView.playertrans, Renderer.players);
  pfAddChild(LeftView.terraintrans, Renderer.terrain->root);
  LeftView.setup_drawing();
      :
      :
  rightView.new_view(0);
  // New code for recovering the lost function because of
  // breaking the mutual dependency.
  Renderer.arbitrate(&RightView,0);
  Renderer.terrain->configure_channel(RightView.chan);
  pfAddChild(RightView.playertrans, Renderer.players);
  pfAddChild(RightView.terraintrans, Renderer.terrain->root);
  RightView.setup_drawing();
      :
      :
}
```

Figure C-6. The necessary change of RDT after breaking the second mutual dependency

Another modification to ObjectSim for breaking this mutual dependency is the derived class of `View`, called `Multiview`. Since the `Pfmr_Renderer` attribute has been removed from `View`, its child class, `Multiview`, is no longer able to refer this attribute. In the source-code file of `Multiview`, one member function, called

draw_mods, did refer to this attribute. All references to Pfmr_Renderer in the source-code file of Multiview class has been removed. As a result of these modifications, ObjectSim applications need to call setdrawmode(OVERLAY_SCREEN) before Multiview::draw_mods is called and then call setdrawmode(NORMAL) after Multiview::draw_mods is called. Since RDT did not refer to the Multiview class, RDT was not modified.

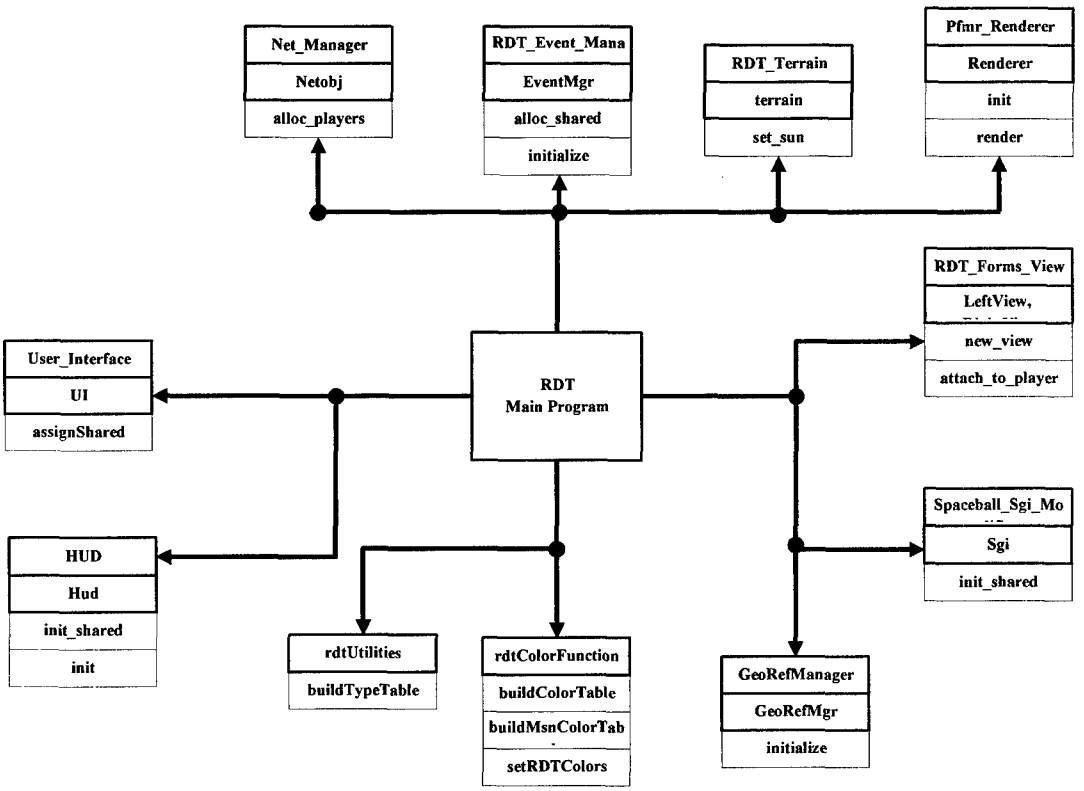
In breaking the second mutual dependency, one lesson is learned. The major difficulty in minimizing the ripple effect is to assure that the relative sequences of the statement and DPs are independent. In some cases, a statement can be moved to any position in a module without any functional change, but other statements must be placed before or after the specified DPs. If software engineers have enough knowledge about the exact sequences of the statements and the DPs, they can rearrange and group the statements as a block and decompose the whole member function into several subfunctions. However, if there is insufficient knowledge about the relative sequencing between the statements and DPs, Figure C-4 presents a way to decide the relative sequence of between statements and DPs and then to minimize the ripple effect while breaking mutual dependencies. In Figure C-4, (2.1) ~ (2.2) and (4.1) ~ (4.2) are the heuristic steps to move DPs closer to each other.

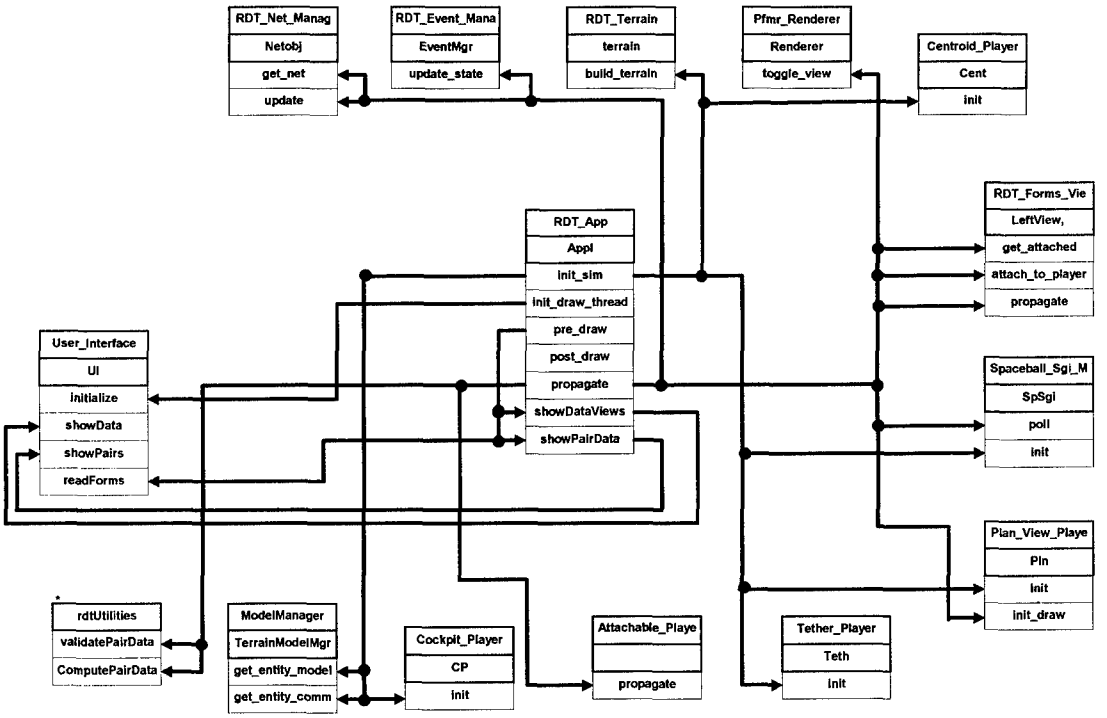
C. 4 Breaking the Third Mutual Dependency

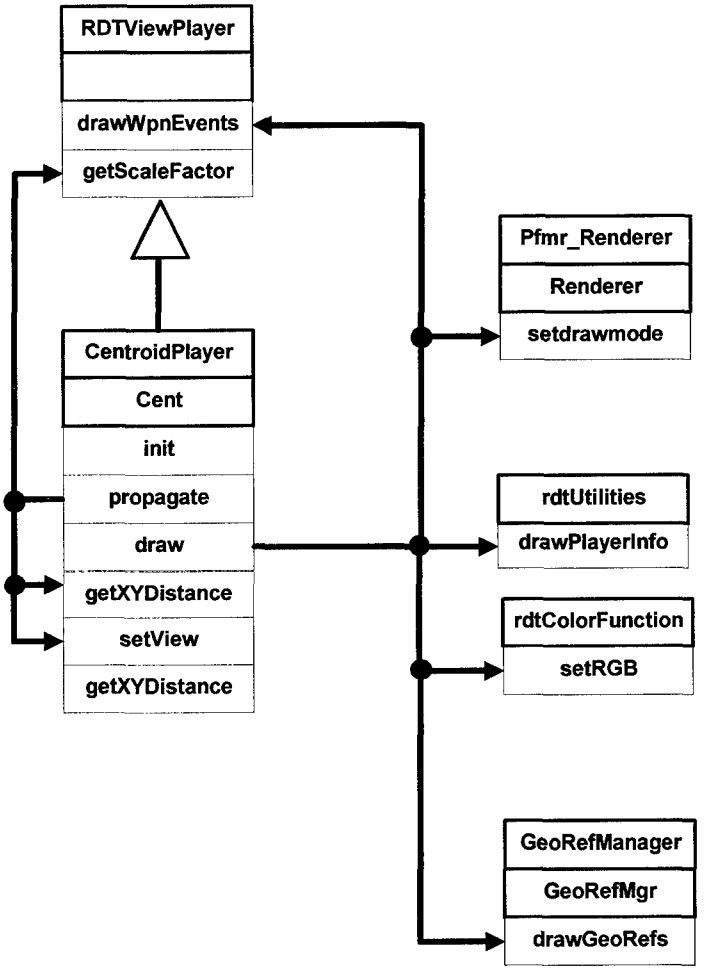
Referring to Figure C-1, the third mutual dependency is a big loop: Player → Pfmr_Renderer → View → Attachable_Player → Player. Based on the same principles and same reasons as was used to break the first mutual dependency, the inclusion of Pfmr_Renderer in the Player class and the associated

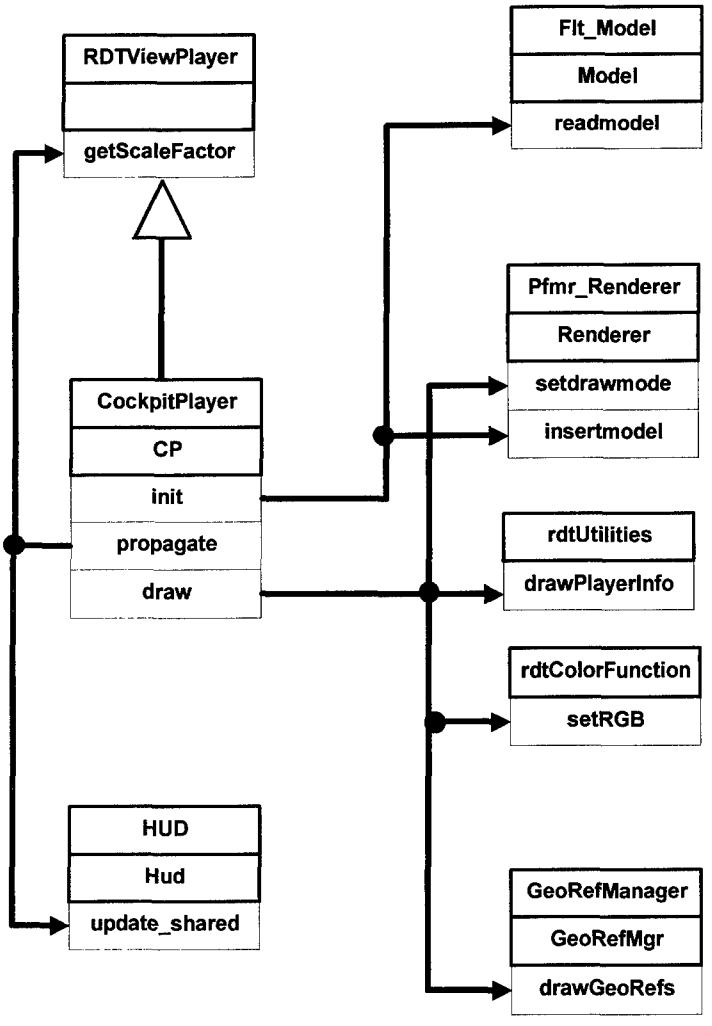
Pfmr_Renderer attribute was removed. Player included Pfmr_Renderer because the header file of Player declared an attribute to associate the Pfmr_Renderer. However, the source-code file of Player never referred to the Pfmr_Renderer. Therefore, only the Pfmr_Renderer attribute in the header file of Player was removed. No change was made in the source-code file of Player.

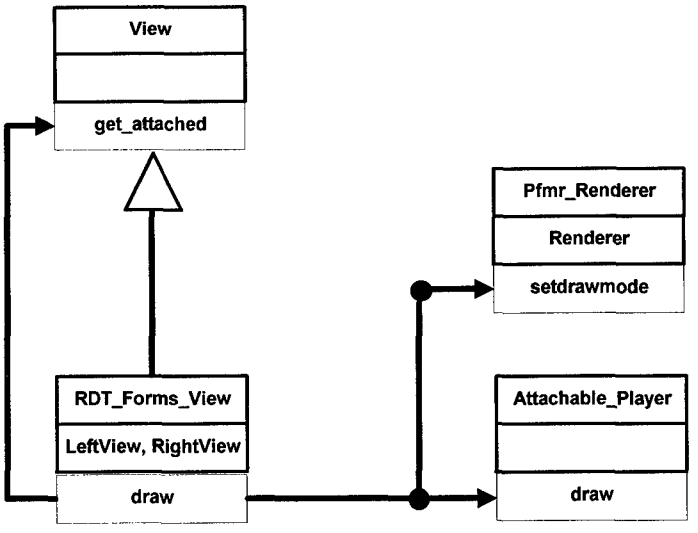
Appendix D. The Calling Dependency Diagrams of RDT

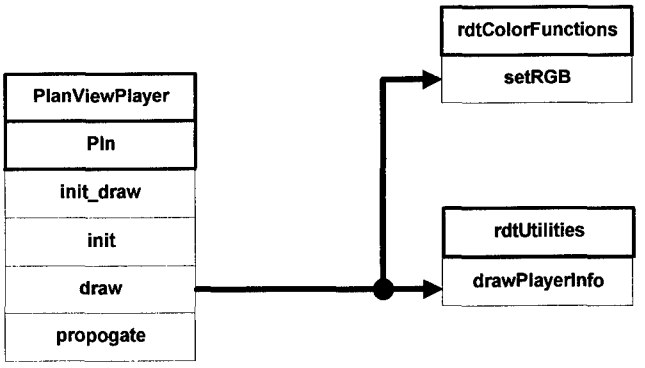


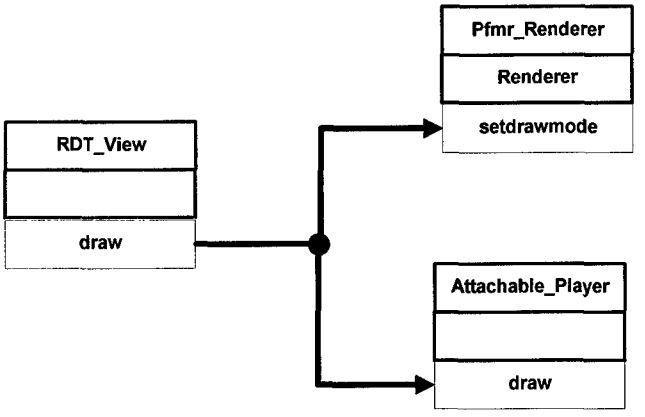

RDT_Net_Manag
Netobj
get_net
update
RDT_Event_Mana
EventMgr
update_state
RDT_Terrain
terrain
build_terrain
Pfmr_Renderer
Renderer
toggle_view
Centroid_Player
Cent
init
RDT_Forms_Vie
LeftView,
get_attached
attach_to_player
propagate
RDT_App
Appl
init_sim
init_draw_thread
pre_draw
post_draw
propagate
showDataViews
showPairData
User_Interface
UI
initialize
showData
showPairs
readForms
Spaceball_Sgi_M
SpSgi
poll
init
Plan_View_Playe
Pln
init
init_draw
rdtUtilities
validatePairData
ComputePairData
ModelManager
TerrainModelMgr
get_entity_model
get_entity_comm
Cockpit_Player
CP
init
Attachable_Playe
propagate
Tether_Player
Teth
init

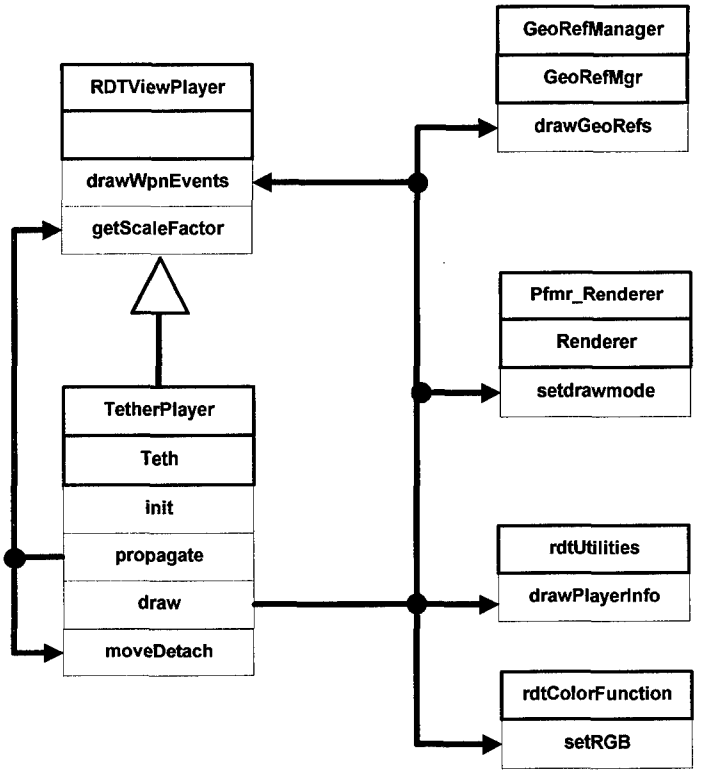

RDTViewPlayer
drawWpnEvents
getScaleFactor
CentroidPlayer
Cent
init
propagate
draw
getXYDistance
setView
getXYDistance
Pfmr_Renderer
Renderer
setdrawmode
rdtUtilities
drawPlayerInfo
rdtColorFunction
setRGB
GeoRefManager
GeoRefMgr
drawGeoRefs

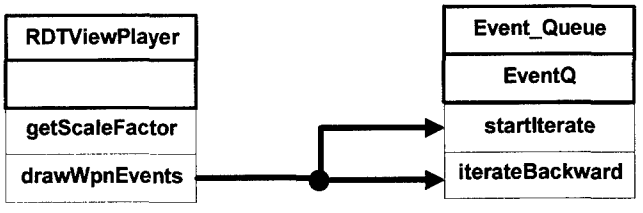

RDTViewPlayer
getScaleFactor
Flt_Model
Model
readmodel
Pfmr_Renderer
Renderer
setdrawmode
insertmodel
CockpitPlayer
CP
init
propagate
draw
rdtUtilities
drawPlayerInfo
rdtColorFunction
setRGB
HUD
Hud
update_shared
GeoRefManager
GeoRefMgr
drawGeoRefs

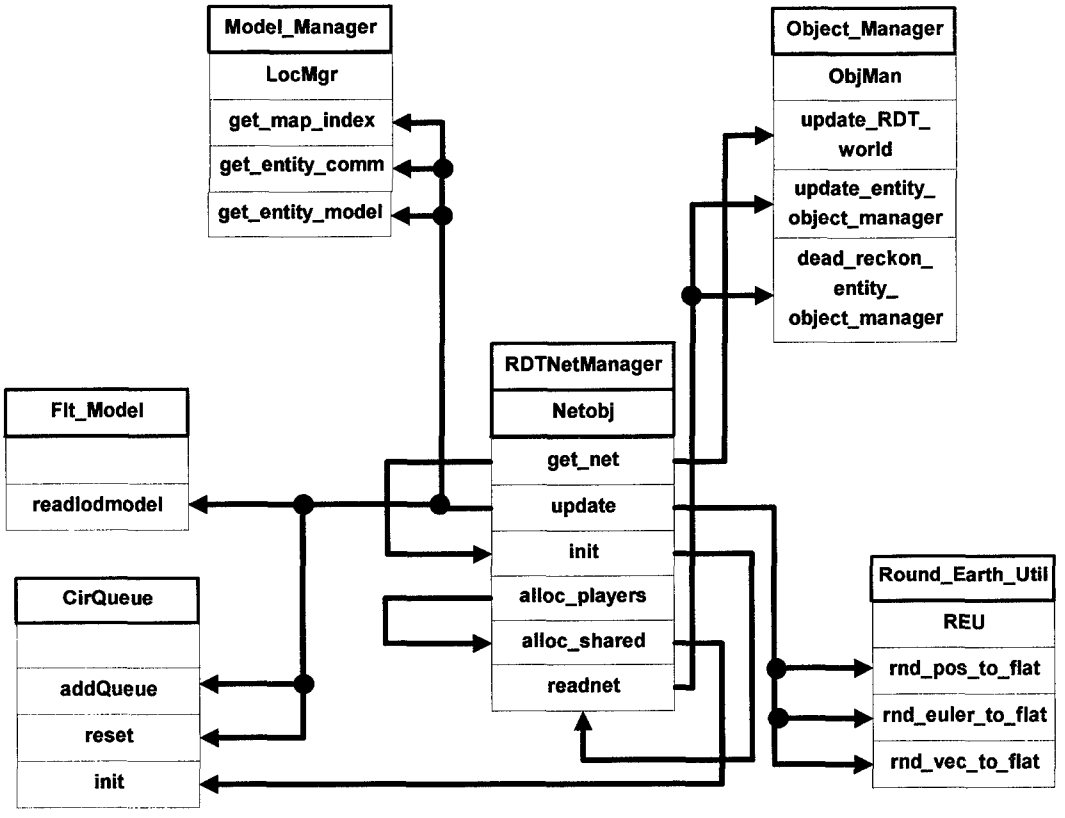

View
get_attached
Pfmr_Renderer
Renderer
setdrawmode
RDT_Forms_View
LeftView, RightView
draw
Attachable_Player
draw

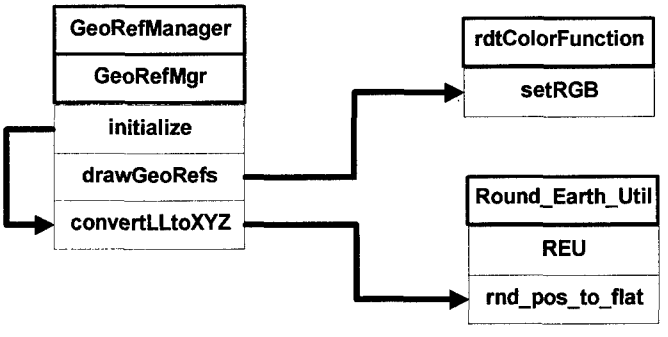

rdtColorFunctions
setRGB
PlanViewPlayer
Pln
init_draw
init
draw
propogate
rdtUtilities
drawPlayerInfo


Pfmr_Renderer
Renderer
setdrawmode
RDT_View
draw
Attachable_Player
draw


GeoRefManager
GeoRefMgr
drawGeoRefs
RDTViewPlayer
drawWpnEvents
getScaleFactor
Pfmr_Renderer
Renderer
setdrawmode
TetherPlayer
Teth
init
propagate
draw
moveDetach
rdtUtilities
drawPlayerInfo
rdtColorFunction
setRGB


RDTViewPlayer
getScaleFactor
drawWpnEvents
Event_Queue
EventQ
startIterate
iterateBackward


Model_Manager
LocMgr
get_map_index
get_entity_comm
get_entity_model
Object_Manager
ObjMan
update_RDT_ world
update_entity_ object_manager
dead_reckon_ entity_ object_manager
RDTNetManager
Netobj
get_net
update
init
alloc_players
alloc_shared
readnet
Flt_Model
readlodmodel
CirQueue
addQueue
reset
init
Round_Earth_Util
REU
rnd_pos_to_flat
rnd_euler_to_flat
rnd_vec_to_flat


GeoRefManager
GeoRefMgr
initialize
drawGeoRefs
convertLLtoXYZ
rdtColorFunction
setRGB
Round_Earth_Util
REU
rnd_pos_to_flat

Appendix E. A Storage_Error Case: Choosing the Wrong Linker

This appendix presents a case where Storage_Error is raised because of choosing the wrong linker. The object model of this example is shown in Figure E-1. The implementations of `Class_A`, `Class_AA`, and `Class_B` are presented in Figure E-2, Figure E-3, and Figure E-4, respectively. In Figure E-2, the constructor of `Class_A` initializes `a_val` to 1111. In Figure E-3, `Class_AA` is a child class of `Class_A` and `a_val` is initialized to 7777. In Figure E-4, `Class_B` has an association with `Class_A`. This association is implemented as a polymorphic member function of `Class_B`, named `B_func_1`, which has a class pointer parameter. This means the parameter could be an object of the `Class_A` or any children of `Class_A`. Figure E-5 shows the main program and its execution results. In the main programs, when `AA_Ptr` is allocated, its `a_val` is initialized to 7777 not 1111 (See the constructor of `Class_AA` in Figure E-3). The right-hand side of Figure E-5 shows the execution results which prove that the polymorphism of `B_func_1` was achieved.

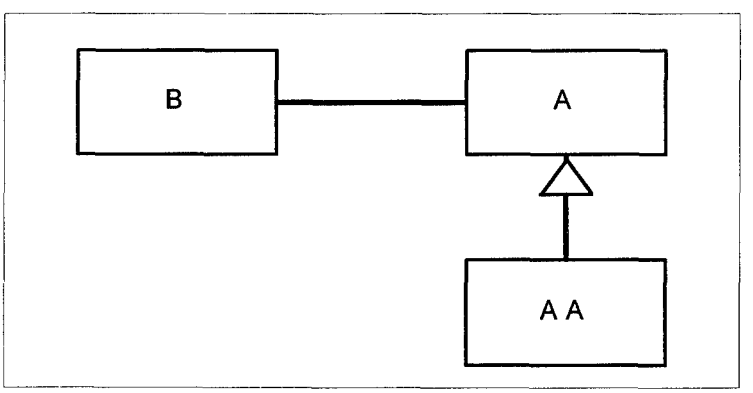


Figure E- 1. The object-oriented model of the programs to interface a C++ polymorphic function

```
// class_A.h
#ifndef __class_A
#define __class_A
class Class_A {
public:
            Class_A();
int         a_val;
virtual     a_fun_1()=0;
};
#endif
```

```
//class_A.cc
#include "class_A.h"
Class_A::Class_A()
{
   a_val = 1111;
}
```

Figure E- 2. The header file and source-code file of Class_A

```
//class_AA.h
#include "class_A.h"
class Class_A;
class Class_AA : public Class_A
{
public:
 int  aa_val;
      Class_AA::Class_AA();
      a_fun_1();
};
```

```
//class_AA.cc
#include "class_AA.h"
Class_AA::Class_AA(void) {
aa_val = 2222;
a_val = 7777;
}
Class_AA::a_fun_1(void) {
a_val = 8888;
}
```

Figure E- 3. The header file and source-code file of Class_AA

```
//class_B.h
#ifndef __class_B
#define __class_B
#include "class_A.h"
class Class_A;
class Class_B {
public:
void B_func_1(Class_A* A); };
#endif
```

```
//class_B.cc
# include "class_B.h"
# include <iostream.h>
void Class_B::B_func_1(Class_A* A)
{
cout << "This is B_func_1\n";
cout << "B->A.a_val ="
     << (*A).a_val << "\n";
}
```

Figure E- 4. The header file and source-code file of Class_B

```
// cpp_main.cc
#include "class_AA.h"
#include "class_B.h"
#include <iostream.h>
void main() {
Class_B         B;
Class_AA*       AA_Ptr;
cout << "This is main program\n";
AA_Ptr = new Class_AA();
cout << "Calling B_func_1\n";
B.B_func_1(AA_Ptr);
cout << "End of the main program\n";
}
```

```
This is main program
Calling B_func_1
This is B_func_1
B->A.a_val=7777
End of the main program
```

Figure E- 5. The C++ main program and its execution results

In order to interface these three existing C++ classes, three Ada package specifications were created: class_A_h, class_AA_h, and class_B_h which are shown in Figure E-7, Figure E-8, and Figure-9 respectively. A non-OOP programmer would produce a profile of `B_func_1` that would look like as follow:

```
procedure B_func_1 (P1: Class_B; A: Class_A_Ptr);
```

However, this does not work in the OOP environment, because directly converting the type of the parameter `A` into `Class_A_Ptr` will not achieve the polymorphism result. Passing a `Class_AA_Ptr` parameter will not be allowed in Ada if `B_func_1` is declared as above. Therefore, in order to achieve the polymorphism of the `B_func_1`, the type of the parameter A should be declared as `Class_A_ClassWide_Ptr`. The profile of `B_func_1` should look as follows:

```
procedure B_func_1( P1: Class_B; A:class_A_ClassWide_Ptr);
```

where `Class_A_ClassWide_Ptr` is declared as the following:

```
type Class_A_ClassWide_Ptr is access all Class_A'Class;
```

Finally, the C++ main program, shown in Figure E-5, is converted to an Ada procedure, called `ex_main` and shown in Figure E-9. In theory, the same result as the right-hand side of Figure E-5 is expected. However, executing `ex_main` raised Storage_Error when `B_func_1` was called. The result is shown as follows:

```
This is main program
Calling B_func_1
raised Storage_Error
```

```
--class_a_h.ads
with Interfaces.CPP;
use  Interfaces.CPP;
package class_A_h is
   type Class_A is abstract tagged
      record
         a_val : Integer;
         vptr  : Interfaces.CPP.Vtable_Ptr;
      end record;
   pragma CPP_Class (Entity => Class_A);
   pragma CPP_Vtable (Entity=>Class_A,Vtable_Ptr=>vptr,Entry_Count=>1);
   type Class_A_ClassWide_Ptr is access all Class_A'class;
   type Class_A_Ptr is access all Class_A;
   function  Constructor  return Class_A'Class;
   pragma CPP_Constructor (Entity => Constructor);
   pragma Import (C, Constructor, "Class_A", "__ct__7Class_AFv");
   function  a_fun_1 (This : Class_A) return Integer;
   pragma CPP_Virtual(Entity=>a_fun_1,Vtable_Ptr=>vptr,Entry_Count=>1);
   pragma Import (C, a_fun_1, "a_fun_1", "a_fun_1__7Class_AFv");

end class_A_h;
```

Figure E- 6. The interface package specification of Class_A

```
-- class_aa_h.ads
with class_a_h;
use  class_a_h;
package class_AA_h is
 type Class_AA is new Class_A with
      record
         aa_val : Integer;
      end record;
   pragma CPP_Class (Entity => Class_AA);
   type Class_AA_Ptr is access all Class_AA;
   function  Constructor  return Class_AA'Class;
   pragma CPP_Constructor (Entity => Constructor);
   pragma Import (C, Constructor, "Class_AA", "__ct__8Class_AAFv");
end class_AA_h;
```

Figure E- 7. The interface package specification of Class_AA

```
-- class_b_h.ads
with class_A_h;
use  class_A_h;
package class_B_h is
  type Class_B is null record;
  procedure  B_func_1 (P1 : Class_B; A : Class_A_ClassWide_Ptr);
  pragma Import(C,B_func_1, "B_func_1","B_func_1__7Class_BFP7Class_A");
end class_B_h;
```

Figure E- 8. The interface package specification of Class_B

```
-- ex_main.adb
with class_b_h;
use  class_b_h;
with class_aa_h;
use  class_aa_h;
with class_a_h;
use  class_a_h;with Text_IO;
use  Text_IO;
procedure ex_main is

B : Class_B;
AA_ptr: Class_AA_Ptr;

begin
   put_line("This is main program");
   AA_ptr := new Class_AA;
   put_line("Calling B_func_1");
   class_b_h.B_func_1(B, Class_A_ClassWide_Ptr(AA_ptr));
   put_line("ENd of the main program");
end ex_main;
```

Figure E- 9. The converted Ada main program

After receiving help from Dr. Comar at New York University, the converted Ada main program, named `ex_main`, correctly works now. The real problem is not related to the programs, but to the wrong choice of linker. Figure E-10 shows the original building steps and Figure E-11 shows the correct building steps. The only difference is in the linker. In the original building steps, gnatlink was chosen to link the programs (In Figure E-10, **gnatbl** command means the binding and linking of GNAT.). The second time, the CC linker was used in the correct building steps.

The bottom line is that the wrong linker may cause Storage_Error at run-time even though the linker may not show any error messages. Therefore, programmers may need to try another linker if they cannot find anything wrong with the programs.

```
% CC -c -g class_A.cc
% CC -c -g class_AA.cc
% CC -c -g class_B.cc
% gcc -c -g class_a_h.ads
% gcc -c -g class_aa_h.ads
% gcc -c -g class_b_h.ads
% gcc -c -g ex_main.adb
% gnatbl -v ex_main.ali -o AdaMain class_A.o class_AA.o class_B.o \
        -L/usr/lib -L/usr/local/lib -lgnat -lC
```

Figure E- 10. The original building procedure for the example programs shown in Figure E-2~E-9

```
% CC -c -g class_A.cc
% CC -c -g class_AA.cc
% CC -c -g class_B.cc
% gcc -c -g class_a_h.ads
% gcc -c -g class_aa_h.ads
% gcc -c -g class_b_h.ads
% gnatbind -x ex_main.ali
% gcc -c b_ex_main.c
% CC -o AdaMain class_A.o class_AA.o class_B.o class_a_h.o class_aa_h.o \
        class_b_h.o ex_main.o  b_ex_main.o -L/usr/lib -L/usr/local/lib -lgnat \
        -L/usr/local/lib/gcc-lib/mips-sgi-irix5.3/2.7.2 -lgcc
```

Figure E- 11. The correct building procedure for the example programs shown in Figure E-2~E-9

Appendix F. Deficiencies of Adabindgen

This appendix lists the deficiencies of Adabindgen. The first four points are extracted from [SGI96] and the others points are obtained during this thesis effort.

1. The original C/C++ comments are excluded from the converted Ada package specification.

2. In certain circumstances, Hex or Octal numeric literals are represented in decimal in the Ada package specification.

3. Most C macros with parametera are not represented in the Ada specification.

4. C unions are currently converted into byte arrays in Ada. They will be converted into a discriminated records with a dummy discriminant.

5. The inline functions cannot be correctly converted into Ada subprograms.

6. Each `#include` directive is supposed to be converted into a **with** clause which refers to the converted package specification from the included header file. However, Adabindgen may not convert each `#include` directive to a **with** clause.

7. Duplicated declarations of some types may exist in different package specifications.

8. In some circumstances, constructors may not be converted to an Ada function with a classwide type return value.

9. In some cases, Adabindgen may convert a C++ class as an Ada tagged type derived from its grandparent class instead of its parent class.

10. Currently, all protected and private data attributes and member functions of a C++ class are converted into public data attributes and subprograms of an Ada package. All visibilities of a C++ class's data attributes and member functions are distorted.

Appendix G. The Converted ObjectSim and RDT Header Files

This appendix lists all Ada package specifications created by this thesis work to interface the existing ObjectSim, ObjectManager, and RDT classes. Table G-1 lists the ObjectManager header files used by RDT and the created Ada package specifications. The ObjectSim header files and the created Ada package specifications to interface ObjectSim classes are listed in Table G-2. The original RDT header files and the created Ada package specifications to interface the RDT classes are listed in Table G-3.

Table G- 1. The created Ada interface package specifications of some ObjectManager header files

C++ Header File Name	Ada Package Specification Name
DIS_v2_RDT_entity_obj_mgr.h	dis_v2_rdt_entity_obj_mgr_h.ads
simulation_clock.h	simulation_clock_h.ads
sim_time.h	sim_time_h.ads
dsi_user.h	dsi_user_h.ads
DIS_v2_entity_obj_mgr.h	dis_v2_entity_obj_mgr_h.ads

Table G- 2. The created Ada interface package specifications of ObjectSim header files

C++ Header File Name	Ada Package Specification Name	Used by RDT	Used by testsim
Fastrak.h	fastrak_h.ads	**Y**	
GraphFont.h	graphfont_h.ads	**Y**	
GraphText.h	graphtext_h.ads	**Y**	
animation_effect_player.h	animation_effect_player_h.ads	**N**	
attachable_player.h	attachable_player_h.ads	**Y**	**Y**
base_net_player.h	base_net_player_h.ads	**Y**	
bldg_player.h	bldg_player_h.ads	**N**	
colors.h	colors_h.ads	**N**	
constants.h	constants_h.ads	**Y**	**Y**
cp_colors.h	cp_colors_h.ads	**N**	
crip_pohl.h	crip_pohl_h.ads	**N**	
dartview.h	dartview_h.ads	**N**	
event.h	event_h.ads	**Y**	
flt_model.h	flt_model_h.ads	**Y**	**Y**
labfont.h	labfont_h.ads	**Y**	
model_mgr.h	model_mgr_h.ads	**Y**	
modifier.h	modifier_h.ads	**N**	**Y**
mouse_mod.h	mouse_mod_h.ads	**N**	
multiview.h	multiview_h.ads	**N**	
nq_keypad_modifier.h	nq_keypad_modifier_h.ads	**N**	
nq_modifier.h	nq_modifier_h.ads	**N**	
nq_spaceball_sgi_mod.h	nq_spaceball_sgi_mod_h.ads	**N**	
pfmr_renderer.h	pfmr_renderer_h.ads	**Y**	
player.h	player_h.ads	**Y**	
polhemus_mod.h	polhemus_mod_h.ads	**N**	
round_earth_utils.h	round_earth_utils_h.ads	**N**	**Y**
sim_entity_mgr.h	sim_entity_mgr_h.ads	**N**	
simple_terrain.h	simple_terrain_h.ads	**Y**	
simulation.h	simulation_h.ads	**Y**	**Y**
spaceball_mod.h	spaceball_mod_h.ads	**N**	
spaceball_sgi_mod.h	spaceball_sgi_mod_h.ads	**Y**	
terrain.h	terrain_h.ads	**Y**	**Y**
vc_net_manager.h	vc_net_manager_h.ads	**N**	
vc_net_player.h	vc_net_player_h.ads	**N**	
view.h	view_h.ads	**Y**	**Y**

Y: Yes. **N: No.**

Table G- 3. The created Ada interface package specifications of RDT header files

C++ Header File Name	Ada Package Specification Name
StringClass.h	stringclass_h.ads
errmsg.h	errmsg_h.ads
flyer.h	flyer_h.ads
hud.h	hud_h.ads
hudtypes.h	hudtypes_h.ads
instruments.h	instruments_h.ads
rdtApplication.h	rdtapplication_h.ads
rdtCentroid.h	rdtcentroid_h.ads
rdtCirQueue.h	rdtcirqueue_h.ads
rdtColorFunctions.h	rdtcolorfunctions_h.ads
rdtConfigure.h	rdtconfigure_h.ads
rdtController.h	rdtcontroller_h.ads
rdtDefines.h	rdtdefines_h.ads
rdtErrorMessages.h	rdterrormessages_h.ads
rdtEvent.h	rdtevent_h.ads
rdtEventManager.h	rdteventmanager_h.ads
rdtEventQueue.h	rdteventqueue_h.ads
rdtGeoRefMgr.h	rdtgeorefmgr_h.ads
rdtGlobals.h	rdtglobals_h.ads
rdtHelp.h	rdthelp_h.ads
rdtNetManager.h	rdtnetmanager_h.ads
rdtPlayer.h	rdtplayer_h.ads
rdtREU.h	rdtreu_h.ads
rdtSetRLE.h	rdtsetrle_h.ads
rdtStrfnct.h	rdtstrfnct_h.ads
rdtTypes.h	rdttypes_h.ads
rdtUserInterface.h	rdtuserinterface_h.ads
rdtUtilities.h	rdtutilities_h.ads
sim_models.h	sim_models_h.ads
wgs84.h	wgs84_h.ads

Appendix H. The Source-code Modules of RDT

RDT C++ Source-code Modules File Name	Converted Ada Package Body File Name*
CentroidPlayer.cc	
CockpitPlayer.cc	
PipeClass.cc	
PlanViewPlayer.cc	
StringClass.cc	
TetherPlayer.cc	
hud.cc	
instruments.cc	
rdtApp.cc	
rdtApplication.cc	rdtMain.adb
rdtCentroid.cc	
rdtCirQueue.cc	rdtcirqueue_h.adb
rdtColorFunctions.cc	
rdtConfigure.cc	
rdtController.cc	
rdtErrorMessages.cc	
rdtEventManager.cc	rdteventmanager_h.adb
rdtEventQueue.cc	
rdtGeoRefMgr.cc	
rdtHelp.cc	
rdtNetManager.cc	
rdtREU.cc	
rdtSetRLE.cc	
rdtTerrain.cc	
rdtUserInterface.cc	
rdtUtilities.cc	
rdtView.cc	
rdtViewPlayer.cc	
wgs84.cc	

* Blank indicates no converted Ada package body.

Appendix I. An Example to Interface an Ada Tagged Type from C++

This appendix shows an example of how to use GNAT's C++ low-level interface capabilities to interface an Ada tagged type from C++ programs. Figure I-1 shows the package specification of an interfaced Ada tagged type, called `Class_A`. In order to let the C++ program interface it, `Class_A` must add an attribute, called `vtr` as a component. Then, the `Class_A`'s member function, `a_fun_1`, must be exported as a C link symbol so that other C++ modules can interface it. The package body of `Class_A` is shown in Figure I-2.

In order to derive a C++ class, called `Class_AA`, from the Ada tagged type, `Class_A`, all data attributes and member functions of `Class_A` must be declared within the declaration of `Class_AA`, like the declaration shown in Figure I-3. In the declaration of `Class_AA`, one data attribute and one member function, `aa_val` and `aa_func_2`, are added into the declaration of `Class_AA`. The data attribute, `a_val`, and the member function, `aa_func_1` are declared in order to interface the original Ada tagged type, `Class_A`. Programmers must make sure that all data attributes and member functions of the parent Ada tagged type are declared in the same sequence as they are declared in the Ada tagged type declaration. Then, new data attributes and member function declarations should follow. For instance, in this example, `a_val` must be declared before `aa_val`, because `a_val` is a data attribute of the original Ada tagged type. For the same reason, the member function `aa_func_1` must be declared before `aa_func_2`.

To implement the member functions of `Class_AA`, the exported original member function of the interfaced Ada tagged type must be specified as a C link symbol. In Figure

I-4, the external declaration section specifies `a_fun_1` as a C link symbol. Then, `a_fun_1` is wrapped in `aa_func_1`. In this way, the original member function `a_fun_1` is indirectly executed.

Figure I-5 is a C++ main program to show that correct interface results are achieved. In the C++ main program, two extra functions must be called. `Adainit` must be called before any declaration and execution statements and `adafinal` must be called after all execution statements. `Adainit` is called to elaborate the Ada library units and `adafinal` is called to perform the finalization that normally takes place after return from an Ada main subprogram [Cohen96].

```
with Interfaces.CPP;
use  Interfaces.CPP;

package class_A_h is

    type Class_A is tagged
      record
        a_val : Integer := 120;
        vptr  : Interfaces.CPP.Vtable_Ptr;
      end record;

    pragma CPP_Class (Entity => Class_A);
    pragma CPP_Vtable (Entity => Class_A, Vtable_Ptr => vptr, Entry_Count => 1);
    function  a_fun_1 (This : Class_A'Class) return Integer;
    pragma Export (C, a_fun_1, "a_fun_1", "a_fun_1");

end class_A_h;
```

Figure I- 1. The package specification of an interfaced Ada tagged type, named Class_A

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use  Ada.Text_IO, ada.Integer_Text_IO;

package body class_A_h is

 function a_fun_1 (This : Class_A'Class) return Integer is
 begin
        put_Line("This is a_fun_1");
        put("a_val = ");
        Put(This.a_val);
        put_line(" ");
        return This.a_val;
 end a_fun_1;
end class_A_h;
```

Figure I- 2. The package body of Class_A

```
class Class_AA
{
public :
  int a_val;
  int aa_val;
  int aa_func_1(void);
  int aa_func_2(void);
     Class_AA();
};
```

Figure I- 3. The declaration of Class_AA

```
#include "class_AA.h"
extern "C" {
 int a_fun_1(void);
}

int Class_AA::aa_func_1(void) {
 return a_fun_1();
}

int Class_AA::aa_func_2(void) {
 return aa_val;
}

Class_AA::Class_AA() {
 a_val =120;
 aa_val=222;
}
```

Figure I- 4. The implementation of Class_AA

```
CppMain: CppMain.cc class_a_h.o b_class_a_h.o class_AA.o
        CC CppMain.cc -o CppMain class_a_h.o b_class_a_h.o class_AA.o \
    -lgnat -L/usr/local/lib/gcc-lib/mips-sgi-irix5.3/2.7.2 -lgcc

class_a_h.o: class_a_h.adb class_a_h.ads
        gcc -c class_a_h.adb

b_class_a_h.o: class_a_h.o b_class_a_h.c
        gnatbind -g -n class_a_h.ali
        gcc -c b_class_a_h.c

class_AA.o: class_AA.h class_AA.cc
        CC -c -g class_AA.cc
clean:
        rm *.o
```

Figure I- 5. The Makefile to build this example

Appendix J. The Conversion of a C++ Function

This appendix presents the rules to convert C++ functions into Ada subprograms. C++ programs consist of pieces called classes and functions [Deitel94]. A C++ class can be converted into an Ada tagged type and a C++ function should be converted into an Ada procedure or function based on the following rules:

1. A C++ function without any return value should be converted into an Ada procedure.

2. A C++ function with a return value should be converted into an Ada function.

For example, a C++ function has the following profile:

```
void function1 (int p1);
```

Then it should be converted into an Ada procedure as follows:

```
procedure function1 (p1 : Integer);
```

Also, a C++ function has the following profile:

```
int function2 (int p1);
```

Then it should be converted into an Ada function as follows:

```
function function2 (p1 : integer ) return Integer;
```

However, there is one exception. If a C++ member function modifies the class data attributes and has a return value, it is mandatory to change it into a C++ class member function without the return value. Then this changed C++ member function should be converted into an Ada procedure. Consider the C++ class and its member function shown in Figure J-1. According to the first rule, `AddItem` member function should be converted

into an Ada procedure. Because the `AddItem` procedure modifies `Count`, which is a data attribute of the class, the parameter mode of `This` should be declared as **in out** mode, like the `AddItem` procedure shown in Figure J-2. For the same reason, the `This` parameter in `DeleteItem` also needs to be declared as **in out** mode. According to the second rule, the C++ `DeleteItem` function should be converted into an Ada function, because it has a return value, like the Ada `DeleteItem` function shown in Figure J-2. However, in Ada programs, a function which has a **out** mode parameter is illegal.

```
Class IntegerQueue
{
 public:
    void AddItem(int Item);
    int   DeleteItem(); // Delete the first item in the queue and return the value
     :
     :
private:
    int   Count=0; //Count how many items are in the queue
     :
     :
};

IntegerQueue::AddItem( int Item)
{
  .........
  Count = Count +1;
}

int IntegerQueue::DeleteItem( )
{
  .........
  Count = Count -1;
}
```

Figure J- 1. A C++ class which has a member function with a return value

```
pacjkage IntegerQueue_h is

 type IntegerQueue is tagged
   record
      :
      :
    Count : Integer := 0; --//Count how many items are in the queue
   end record;

   procedure  AddItem  (This : in out IntegerQueue, Item : Integer);

   function  DeleteItem (This : in out IntegerQueue)  return Integer;
     :
     :
end IntegerQueue_h;

package body IntegerQueue_h is

   procedure AddItem( This : in out IntegerQueue, Item : Integer)  is
   begin
      :
      :
      This.Count := This.Count +1;
      :
   end AddItem;

   function  DeleteItem( This : in out IntegerQueue)  return Integer is
   begin
      :
      :
      This.Count := This.Count - 1;
      return XXX;
      :
   end DeleteItem;

end IntegerQueue_h;
```

Figure J- 2. An example showing that a C++ class member function cannot be directly converted into an Ada function

To overcome this defect, it is recommended that a C++ member function, which has a return value and also modifies any data attributes of the class, should be changed into a C++ member function without any return value. Then the changed C++ member function should be converted into an Ada procedure according to the first rule. It is simple to change a C++ class member function, which has a return value, into a C++ member function

without a return value. For example, the `DeleteIem` function shown in Figure J-1, can be changed into a function which looks like the following function profile:

```
void DeleteItem( int &Result);
```

where `Result` is the return value of `DeleteItem`. Using this method can reduce ripple effects. After that, the `DeleteItem` function can be converted into an Ada procedure as follows:

```
procedure DeleteItem ( This   : in out IntegerQueue,
                       Result : access Integer);
```

(Note: Simply changing the profile of `DeleteItem` into

```
void DeleteItem( int Result);
```

will not achieve the same result, because *pass-by-value* is used in C++ for passing function parameters. "Under pass-by-value, the function never accesses the actual arguments of the call. The values the function manipulates are its own local copies" [Lippman91].)

However, this solution has one disadvantage. The original C++ member function profile can directly return the result, but the converted Ada subprogram must access the return result in an indirect way, because the return value type becomes an access type. This loss should be recovered in the last phase of this methodology.

The recovering method is to first change the `Result` type of the Ada procedure from an access type into the original type. Then, some necessary changes in the Ada procedure body must be done, because the calling interface has been changed. Finally, all statements calling the Ada procedure should be changed to meet the new calling interface.

For example, the `DeletItem` procedure should be changed into an Ada procedure whose profile should look like this:

```
procedure DeleteItem ( This   : in out IntegerQueue,
                       Result : in out Integer);
```

Then, the second step is to change the body of the `DeleteItem` procedure. Finally, all statements calling the `DeleteItem` must do the necessary changes to fit the new procedure profile.

In this thesis work, only two RDT class member functions, `EventQueue::iterateBackward` and `CircleQueue::iterateBackward`, must be changed into member functions which have no return values. The new member function profiles are listed in Table J-1.

Table J- 1. The original and new function profiles of CircleQueue::iterateBackward and EventQueue::iterateBackward

Original function profile	New function profile
int CircleQueue::iterateBackward(void)	void CircleQueue::iterbackward(int &Result)
int EventQueue::iterateBackward(void)	void EventQueue::iterbackward(int &Result)

Vita

Capt. Ding-yuan Sheu [redacted], [redacted] He majored in Applied Mathematics at Chung-Cheng Institute of Technology (CCIT) in 1983. After he graduated in 1987, he entered active duty in the Taiwan Army and served in Chung-Shan Institute of Science and Technology (CSIST), Department of Defense, ROC for 7 years. At that time, he was assigned to develop software for network communications. He entered the Air Force Institute of Technology, USA, in May, 1994.

[redacted]

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE
An Incremental Language Conversion Method to Convert C++ to Ada95

5. FUNDING NUMBERS

6. AUTHOR(S)
Ding-yuan Sheu, Captain, ROC Army

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
2750 P Street
WPAFB, OH 45433-7126

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/96D-33

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
ATTN: Mr. Gary Shupe
DISA/DVSW/JEXSV
5600 Columbia Pike
Falls Church, VA 22041

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This thesis develops a methodology to incrementally convert a legacy object-oriented C++ application into Ada95. Using the experience of converting a graphic application, called Remote Debriefing Tool (RDT), in the Graphics Lab of the Air Force Institute of Technology (AFIT), this effort defined a process to convert a C++ application into Ada95.

The methodology consists of five phases: (1) reorganizing the software application, (2) breaking mutual dependencies, (3) creating package specifications to interface the existing C++ classes, (4) converting C++ code into Ada programs, and (5) embellishing. This methodology used the GNAT's C++ low-level interface capabilities to support the incremental conversion. The goal of this methodology is not only to correctly convert C++ code into Ada95, but also to take advantage of Ada's features which support good software engineering principles.

14. SUBJECT TERMS	15. NUMBER OF PAGES
C++, Ada95, Language conversion, Object-oriented, Mixed language programming, Incremental conversion, Software engineering	146
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- **DOD** - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- **DOE** - See authorities.
- **NASA** - See Handbook NHB 2200.2.
- **NTIS** - Leave blank.

Block 12b. Distribution Code.

- **DOD** - Leave blank.
- **DOE** - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- **NASA** - Leave blank.
- **NTIS** - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.